Tällaisia ovat esim. Boost-kirjaston `shared_ptr` [Boost, 2003] tai Andrei Alexandrescun Loki-kirjaston uskomattoman monipuolinen `SmartPtr` [Alexandrescu, 2001] [Alexandrescu, 2003].

11.8 Olio-ohjelmointi ja poikkeusturvallisuus

Tähän mennessä tässä luvussa on käsitelty C++:n poikkeusmekanismin perusteet. Vaikka itse mekanismi ei olekaan kovin monimutkainen, on vikasietoisen ja poikkeuksiin varautuvan ohjelman kirjoittaminen kuitenkin yleensä erittäin monimutkaista ja tarkkuutta vaativaa työtä. Syynä tähän on, että virhetilanteita – ja näin ollen myös poikkeuksia – voi tapahtua lähes missä tahansa kohdassa ohjelmaa.

Olio-ohjelmoinnin kapselointi piilottaa luokkien sisäisen toteutuksen, joten luokan käyttäjä ei voi nähdä, miten luokka on toteutettu ja millaisia virheitä koodissa voi syntyä. Kapselointi tekee myös mahdolliseksi sen, että luokan toteutusta muutetaan myöhemmin, jolloin uuden toteutuksen reagointi virheisiin voi poiketa aiemmasta. Kaiken kukkuraksi periytyminen ja polymorfismi aiheuttavat sen, ettei luokkahierarkian käyttäjä edes välttämättä tarkasti tiedä, minkä luokan oliota käyttää (kun olio on kantaluokkaosoittimen päässä).

Kaikki tämä tekee entistä tärkeämmäksi sen, että kaikki mahdolliset luokasta tai moduulista ulos vuotavat virhetilanteet ja poikkeukset dokumentoidaan rajapinnan dokumentaatiossa. Näin luokan käyttäjä voi varautua kaikkiin tarpeellisiin poikkeustilanteisiin ilman, että hän tietää luokan tai moduulin sisäistä toteutusta.

Samoin periytymishierarkiassa on tärkeää, että aliluokan uudelleen määrittelemät virtuaalifunktiot eivät aiheuta sellaisia virhetilanteita ja poikkeuksia, joita ei ole dokumentoitu jo kantaluokan rajapintadokumentaatiossa. Luokan rajapinnasta vuotavat poikkeustilanteet ja luokan reagoiminen niihin kuuluvat suoraan periytymisen “aliluokan olio on myös kantaluokan olio” -suhteeseen, joten aliluokan tulee noudattaa kantaluokan käyttäytymistä myös poikkeustilanteissa. Toisaalta tämä tarkoittaa myös sitä, että kantaluokka ei saa omassa rajapintadokumentaatiossaan tarjota liian suuria lupauksia poikkeustilanteissa, koska tällöin saattaa pahimmassa tapauksessa käydä niin, että on mahdotonta kirjoittaa aliluokkaa, jonka laajennettu ja muutettu toiminnallisuus edelleen pitäisi kaikista kantaluokan lupauksista kiinni.

Virhetilanteissa järkevästi toimivien ja vikasietoisten luokkien kirjoittaminen tulee helpommaksi, jos ensin määritellään joukko pelisääntöjä, joita olioiden tulee virhetilanteissa noudattaa. Lisäksi olioiden käyttäytyminen virhetilanteissa voidaan jakaa selkeisiin kategorioihin, jolloin rajapintadokumentaation kirjoittaminen ja ymmärtäminen tulee helpommaksi. Tässä aliluvussa esitellään C++:ssa usein käytettävät poikkeusturvallisuuden tasot sekä muutamia yleisiä poikkeuksiin ja C++:n luokkiin liittyviä mekanismeja.

11.8.1 Poikkeustakuut

C++:n poikkeuksista ja niiden käytöstä on jo ehtinyt kertyä käytännön kokemusta, vaikka poikkeusmekanismi tulikin kieleen varsin myöhäisessä vaiheessa. Lisäksi vikasietoisuudesta on tietysti paljon tietämystä myös ajalta ennen C++:aa. Mistään loppuunkalutusta aiheesta ei kuitenkaan ole kysymys, vaan uusia poikkeuksiin liittyviä mekanismeja ja koodaustapoja kehitetään jatkuvasti.

Periaatteessa luokan pitäisi erikseen dokumentoida jokaisesta palvelustaan, mitä virhetilanteita palvelussa voi sattua, ja miten palvelu reagoi niihin. Tämä ei kuitenkaan ole aina järkevää, koska tällöin rajapintadokumentti saattaa helposti kasvaa niin suureksi, että sen käyttökelpoisuus vaarantuu. Lisäksi luokka ei aina voi edes tarkasti määritellä kaikia mahdollisia virhetilanteita. Tätä esiintyy sitä enemmän, mitä geneerisempi ja yleiskäyttöisempi luokka on.

Esimerkiksi C++:n `vector` ei voi millään luetella `push_back`-operaatiosta ulos vuotavia poikkeuksia. Kyseinen operaatiohan aiheuttaa uuden alkion luomisen ja lisäämisen vektoriin, ja vektorilla ei ole mitään käsitystä siitä, millaisia virhetilanteita uuden alkion luomiseen voi liittyä.

Tällaisten ongelmien ratkaisemiseksi voidaan antaa yksinkertainen jaottelu siitä, miten luokka voi tyypillisesti virhetilanteisiin suhtautua. Tässä esitellään tämän jaottelun perusteet, tarkemmin asiaan voi tutustua esim. kirjoista “Exceptional C++” [Sutter, 2000] ja “More Exceptional C++” [Sutter, 2002c].

Alla olevan jaottelun poikkeuksiin suhtautumisesta on ilmeisesti ensimmäisenä julkaissut David Abrahams. Hänen mukaansa luokka voi tarjota operaatioilleen erilaisia **poikkeustakuita** (*exception guarantee*). [Abrahams, 2003]

Minimitakuu (*minimal guarantee*)

Vähin, mitä luokka voi tehdä, on taata, että mikäli olion palvelu keskeytyy virhetilanteen vuoksi (ja poikkeus vuotaa ulos), niin *olio ei hukkaa resursseja ja on edelleen sellaisessa tilassa, että sen voi tuhota.* Tämä tarkoittaa, että virhetilanteenkaan sattuessa olio ei aiheuta muistivuotoja eikä muitakaan resurssivuotoja. Olion ei tarvitse sisäisesti olla "järkevässä" tilassa (luokkainvariantin ei tarvitse olla voimassa), mutta sen purkajan tulee pystyy hoitamaan tarvittavat siivoustoimenpiteet.

Minimitakuu takaa siis vain, että olion voi tuhota ilman ongelmia. Mahdollisesti lisäksi olion "resetoiminen" sopivalla jäsenfunktiolla voi olla mahdollista, tai uuden arvon sijoittaminen olioon.

On varsin selvää, että minimitakuuta löyhempää lupausta ei ole käytännöllistä antaa. Jos oliota ei voi edes turvallisesti tuhota virheen jälkeen, ja se voi vuotaa muistia ja resursseja, ei olion käyttämistä saa turvalliseksi millään keinoin.

Perustakuu (*basic guarantee*)

Perustakuu takaa kaiken minkä minimitakuukin, mutta lisäksi se takaa, että olion luokkainvarianttia ei ole rikottu. Olio on siis poikkeuksen jälkeenkin käyttökelpoisessa tilassa, ja sen jäsenfunktioita voi kutsua. On kuitenkin huomattava, ettei perustakuu tarkoita sitä, että olion täytyisi olla *ennustettavassa* tilassa virhetilanteen jälkeen. Perustakuu takaa vain, että olio ei ole mennyt rikki poikkeuksen johdosta. Olion tila voi olla ennallaan, puolivälissä kohti onnistunutta suoritusta tai olio voi olla muuttunut johonkin aivan toiseen lailliseen tilaan.

Jos esimerkiksi vektorin `insert`-operaatiolla vektoriin lisätään useita alkioita, ja operaation aikana tapahtuu virhe (esim. alkion kopioiminen vuotaa poikkeuksen), ei vektorin alkioista enää ole varmuutta. Saattaa olla, että osa alkioista jää väärään paikkaan vektorissa tai jotkin alkiot saattavat jopa olla vektorissa kahteen kertaan tai puuttua kokonaan. Siitä huolimatta tiedetään, että vektorin voi edelleen tuhota normaalisti, sen voi tyhjentää, ja sen alkiot voi edelleen käydä läpi, vaikka alkioiden arvoista ei olekaan varmuutta. `C++`:n standardikirjaston luokat antavat perustakuun lähes kaikista operaa-

tioistaan. Joistain operaatioista luvataan lisäksi vielä enemmän kuin perustakuu vaatii.

Vahva takuu (*strong guarantee*)

Vahva takuu takaa, että kun luokan oliolle suoritetaan jokin operaatio, niin operaatio saadaan joko suoritettua loppuun ilman virheitä, tai *poikkeuksen sattuessa olion tila pysyy alkuperäisenä.* Tämä tarkoittaa sitä, että jos operaatiossa tapahtuu virhe, niin poikkeuksen vuotamisen jälkeen olio on täsmälleen samassa tilassa kuin ennen koko operaatiotakin. Tästä käytetään myös usein englanninkielistä termiä "*commit or rollback*" – operaatio saa joko toimintansa loppuun tai "kierähtää takaisin" tilaansa ennen operaatiota.

Vahvan takuun antavat operaatiot ovat luokan käyttäjän kannalta varsin helppoja, koska virheen sattuessa olion tila on säilynyt ennallaan. Sen sijaan luokan toteuttajalle vahva takuu tuottaa yleensä jonkin verran lisävaivaa. Vahvan takuun voi toteuttaa esimerkiksi niin, että operaation yhteydessä olion tilan tarvittavat osat kopioidaan muualle, ja itse operaatio suoritetaankin tälle kopiolle. Jos operaatio onnistui, voidaan lopputulos sitten siirtää takaisin olioon. Virheen sattuessa varsinaista oliota ei taas olekaan vielä muutettu, joten operaatio voi yksinkertaisesti tuhota työkopion ja heittää poikkeuksen. Aliluvussa 11.9 on esimerkki vahvan takuu tarjoavasta sijoitusoperaattorista.

Vahvan takuun toteuttaminen saattaa usein vaatia luokan koodaamista siten, että se kuluttaa hieman enemmän muistia ja toimii hieman hitaammin kuin ilman takuuta. Sen vuoksi vahva takuu ei olekaan mikään "ihanne", johon tulisi *aina* pyrkiä, mutta joissain tilanteissa se tekee luokan käyttäjän elämän paljon helpommaksi. Aivan kaikkia operaatioita ei lisäksi edes voi kirjoittaa niin, että ne tarjoaisivat vahvan takuun.

C++:n standardikirjasto pyrkii tarjoamaan vahvan poikkeustakuun sellaisille operaatioille, joissa takuu on mahdollista toteuttaa järkevällä vaivalla ja joissa vahvasta takuusta on käyttäjälle eniten hyötyä. Esimerkiksi kaikkien STL:n säiliöiden `push_back`-operaatiot antavat vahvan takuun – jos alkion lisääminen epäonnistuu, sisältää säiliö poikkeuksen vuotaessa samat alkiot kuin ennen operaatiota.

Nothrow-takuu (*nothrow guarantee*)

Kaikkein vahvin poikkeustakuu “nothrow” on varsin yksinkertainen. Se takaa, että operaation suorituksessa *ei voi sattua virheitä*. Mitään poikkeuksia ei siis voi vuotaa operaatiosta ulos, ja operaatio onnistuu aina. Nothrow-takuu on käyttäjän kannalta ideaalinen, koska virheisiin ei tarvitse varautua lainkaan. Sen sijaan on tietysti selvää, että suuri osa operaatioista ei millään voi tarjota nothrow-takuuta, koska niihin sisältyy aina jokin virhemahdollisuus.

Nothrow-takuu on kuitenkin hyödyllinen virheturvallisen ohjelmoinnin kannalta. Joissain tapauksissa nimittäin ohjelmaa ei voi kirjoittaa virheturvalliseksi, elleivät *jotkin* tietyt operaatiot tarjoa nothrow-takuuta. Esimerkiksi vahvan takuun tarjoaminen vaatii, että onnistuneen operaation lopussa työkopion kopioiminen takaisin itse olioon ei voi epäonnistua – kopioimisen täytyy siis tarjota nothrow-takuu. Samoin aliluvussa 11.7 käsitelty automaattiosoitin `auto_ptr` tarjoaa nothrow-takuun suurimmalle osalle operaatioistaan. Lisäksi nothrow-takuun antavat STL:n säiliöiden `erase`, `pop_back` ja `pop_front`.

C++:n poikkeusmekanismin kannalta nothrow-takuu vastaa poikkeusmääreen **`throw()`** käyttöä. Kyseisen poikkeusmääreen käyttäminen ei kuitenkaan ole mitenkään välttämätöntä, vaan nothrow-takuun voi tarjota myös perinteisesti dokumentoimalla.

Poikkeusneutraalius (*exception neutrality*)

Poikkeusneutraalius ei ole vaihtoehtoinen poikkeustakuu perustakuun, vahvan takuun ja nothrow-takuun rinnalla, mutta se liittyy kuitenkin olennaisesti samaan aiheeseen “toisella akselilla”. Poikkeusneutraaliutta on, että yleiskäyttöisen luokan operaatiot vuotavat sisällään olevien komponenttien poikkeukset ulos muuttumattomina. Tämä tarkoittaa lähinnä sitä, että luokka ei itse muuta poikkeuksia jonkin toisen tyyppisiksi, vaan päästää alkuperäisen poikkeuksen käyttäjälle saakka. Poikkeusneutraaliuden lisäksi luokan pitäisi tietysti tarjota myös jokin muista poikkeustakuista, lähinnä joko perustakuu tai vahva takuu. Siten luokka voi tietysti ottaa poikkeuksen väliaikaisesti kiinni ja reagoida siihen (esim. toteuttaakseen vahvan poikkeustakuun).

Hyvä esimerkki poikkeusneutraaliudesta on `vector`. Vektorin poikkeusneutraalius tarkoittaa, että jos esimerkiksi vektorin `push_back`-operaation yhteydessä lisättävän alkion kopioiminen aiheuttaa poikkeuksen (eli alkiotyypin kopiorakentaja vuotaa poikkeuksen), niin vektori tarvittavan siivouskoodin suorittamisen jälkeen vuotaa tämän *saman poikkeuksen* edelleen ulos käyttäjälleen. (Siis käytännössä tämä tapahtuu suorittamalla siivouskoodin lopussa komento **`throw;`**.)

Poikkeusneutraalius on vektorin tapaisten yleiskäyttöisten luokkamallien tapauksessa varsin toivottavaa. Vektorin koodihan ei voi tietää mitään vektorin alkioiden käyttäytymisestä ja niissä mahdollisesti sattuvista virhetilanteista, koska vektorin koodi on täysin alkiotyypistä riippumatonta. Sen sijaan vektorin käyttäjällä on tavallisesti tarkka tieto siitä, miten vektorin alkioissa tapahtuviin virheisiin tulisi reagoida. Tämän vuoksi on tärkeää, että vektori välittää alkioiden poikkeukset käyttäjälleen saakka, jotta poikkeus saadaan käsiteltyä asianmukaisesti.

11.8.2 Poikkeukset ja rakentajat

Olion luominen on sen elinkaaren kannalta erikoinen tapahtuma, koska luomisen onnistumisesta riippuu, onko koko olio olemassa vai ei. Tämän vuoksi luomiseen liittyy poikkeuksien ja virhetilanteiden kannalta joitain hieman erikoisia piirteitä, jotka on syytä käydä läpi.

Olion luomisen vaiheet

Yksi oleellinen kysymys on, *milloin* olio varsinaisesti syntyy, eli milloin alustustoimet ovat niin pitkällä, että voidaan puhua jo “uudesta oliosta”. C++:n oliomalli määrittelee tämän niin, että olion katsotaan lopullisesti syntyneen, kun *kaikki* olion luomiseen kuuluvat rakentajat on suoritettu onnistuneesti loppuun. Tähän kuuluvat niin kantaluokkien rakentajat kuin myös olion jäsenmuuttujien rakentajat, jos jäsenmuuttujat ovat olioita.

Oleelliseksi tämä “syntymishetki” tulee silloin, kun olion luomisen aikana tapahtuu virhe. Jos nimittäin olion luomisen jossain osavaiheessa tapahtuu poikkeus, jonka kyseinen osa päästää vuotamaan ulos, ei oliota ole saatu luotua onnistuneesti. Osa siitä on kuitenkin

todennäköisesti saatu luotua loppuun saakka, kuten esimerkiksi jotkin jäsenmuuttujat ja kenties osa olion kantaluokkaosista.

Jos virhe on kuitenkin tapahtunut ennen kuin kaikki olioon liittyvät rakentajat on saatu suoritettua, ei oliota ole C++:n kannalta saatu luoduksi. Tällöin C++ pitää automaattisesti huolen siitä, että kaikki ne olion osat, jotka on jo ehditty luoda, tuhotaan automaattisesti osana poikkeuskäsittelyä. Tähän kuuluu niin tarvittavien purkajien kutsuminen kuin muistin vapauttaminenkin. Poikkeuksen tekevät tuttuun tapaan dynaamisesti luodut oliot, joita ei tässäkään tapauksessa tuhota automaattisesti. Lopputulos olion luomisen kannalta joka tapauksessa on, että *olio ei koskaan ehtinyt syntyä.*

Listaus 11.10 näyttää esimerkin oliosta, jossa on jäsenmuuttujina useita toisia olioita. Jos nyt henkilöoliota luotaessa syntymäpäivän luominen onnistuu, mutta nimen luomisessa tapahtuu virhe (esim. muisti loppuu), niin henkilöolion luominen keskeytetään. Lisäksi koska syntymäpäivä on jo saatu luotua, niin se tuhotaan automaattisesti.

Dynaamisesti luodut osaoliot

Koska jäsenmuuttujien ja muiden osaolioiden tuhoaminen poikkeuksen sattuessa tapahtuu automaattisesti, ei siitä yleensä aiheudu vaivaa ohjelmoijalle. Sen sijaan dynaamisesti luotujen olioiden kanssa tulee olla erittäin huolellinen. Niitä ei tuhota automaattisesti, joten on erittäin tärkeää, *että olioita ei luoda dynaamisesti rakentajan alus-*

```
1  class Henkilo
2  {
3  public:
4    Henkilo(int p, int k, int v, std::string const& nimi,
5              std::string const& hetu);
6    ~Henkilo();
         ⋮
7  private:
8    Paivays syntymapvm_;
9    std::string nimi_;
10   std::string* hetup_;
11 };
```

LISTAUS 11.10: Esimerkki luokasta, jossa on useita osaolioita

tuslistassa. Jos näin nimittäin tehdään, dynaamisesti luodut oliot jäävät tuhoamatta, jos jonkin niiden jälkeen alustettavan jäsenmuuttujan luominen epäonnistuu. Tällaiseen virheeseen ei voi reagoida edes rakentajan rungon koodissa olevalla virhekäsittelijällä, koska rakentajan koodiin siirrytään vasta, kun kaikki jäsenmuuttujat on onnistuneesti luotu. Jäsenmuuttujan luomisvirheessä rakentajan runkoon ei siis päästä ollenkaan.

Tämän vuoksi kannattaa noudattaa periaatetta, jossa jäsenmuuttujaosoittimet alustetaan rakentajan alustuslistassa nolliksi ja oliot luodaan dynaamisesti niiden päähän vasta rakentajan rungossa. Näin dynaamisesti luotavat oliot luodaan vasta, kun kaikki normaalit jäsenmuuttujat on saatu onnistuneesti luotua. Jos rakentajan rungossa luodaan dynaamisesti useita olioita, täytyy koodissa olla tietysti tarvittava virheenkäsittely, joka varmistaa että virheen sattuessa jo luodut oliot tuhotaan. Listaus 11.11 näyttää esimerkkinä listauksen 11.10 rakentajan toteutuksen.

Automaattiosoittimia käytettäessä tätä ongelmaa ei pääse syntymään, koska virheenkin sattuessa automaattiosoittimen purkaja tuhoaa dynaamisesti luodun olion. Tämän vuoksi automaattiosoittimen päähän voi alustaa dynaamisesti luodun olion jo alustuslistassa.

```
1  Henkilo::Henkilo(int p, int k, int v, std::string const& nimi,
2                   std::string const& hetu)
4    : syntymapvm_(p, k, v), nimi_(nimi), hetup_(0)
5  {
6    try
7    {
8      hetup_ = new std::string(hetu);
9    }
10   catch (...)
11   { // Tänne päästään, jos hetun luominen epäonnistuu
12     // Siivotaan tarvittaessa, olion luominen epäonnistui
13     throw; // Heitetään virhe edelleen käsiteltäväksi
14   }
15 }
```

LISTAUS 11.11: Olioiden dynaaminen luominen rakentajassa

Luomisvirheisiin reagoiminen

Jos olion luomisen yhteydessä tapahtuu virhe esimerkiksi jäsenmuuttujaa luotaessa, ei luotava olio voi millään toipua tästä virheestä, vaan koko olion luominen epäonnistuu. Tällöin jo luotujen osaolioiden purkajia kutsutaan ja poikkeus vuotaa koodiin, jossa olio luotiin. Jos luokassa kuitenkin on tarve virheen sattuessa yrittää toipua virheestä ja kenties yrittää epäonnistuneen osaolion luomista uudelleen, on tähän yksi keino. Sellaiset osaoliot, joiden luominen voi epäonnistua, voi nimittäin luoda tavallisen jäsenmuuttujan sijaan dynaamisesti osoittimen päähän. Tällöin osaolion luomisen **`new`**'llä voi tehdä rakentajan rungossa, jonka virheenkäsittelykoodi sitten yrittää luomista tarvittaessa uudelleen.

Vaikka tavallisten jäsemuuttujien ja kantaluokkaosien luomisvirheistä ei voikaan toipua, saattaa luokalla olla kuitenkin tarve reagoida tällaisiin virheisiin. Kenties luokan tulee kirjoittaa tieto epäonnistumisesta virhelokiin tai antaa käyttäjälle virheilmoitus. Joskus osaolion luomisesta aiheutunut poikkeus voi myös olla väärää tyyppiä, ja luokka haluaisi itse vuotaa ulos toisentyyppisen poikkeuksen.

Standardoinnin myötä C++:aan lisättiin näitä tarpeita varten erityinen **funktion valvontalohko** (*function try block*). Sitä voi käyttää rakentajissa (ja purkajissa), ja sen virhekäsittelijät ottavat kiinni poikkeukset, jotka tapahtuvat jäsenmuuttujien tai kantaluokkaosien luomisen (purkajan tapauksessa tuhoamisen) yhteydessä. Listaus 11.12 seuraavalla sivulla näyttää tämän valvontalohkon syntaksin. Avainsana **`try`** tulee jo ennen rakentajan alustuslistaa, eikä sen jälkeen tule aaltosulkuja. Valvontalohko kattaa automaattisesti virheet, jotka syntyvät osaolioita luotaessa tai itse rakentajan rungossa. Valvontalohkoon liittyvät virhekäsittelijät tulevat aivan rakentajan loppuun sen rungon jälkeen.

Funktion valvontalohkosta on huomattava, että sen virhekäsittelijöihin päästäessä olion ennen virhettä luodut jäsenmuuttujat ja kantaluokkaosat *on jo tuhottu.* Virhekäsittelijöissä ei siis enää voi viitata jäsenmuuttujiin tai kantaluokan palveluihin eikä näin ollen voi suorittaa mitään varsinaisia siivousoperaatioita (kuten dynaamisen muistin vapauttamista). Samoin virhekäsittelijä *ei voi käsitellä virhettä loppuun ja näin toipua siitä*, vaan jokaisen virhekäsittelijän on lopuksi joko heitettävä sama virhe uudelleen (komennolla **`throw;`**) tai sitten heitettävä kokonaan uusi erityyppinen virhe. Jos virhekäsitteli-

```
1  Henkilo::Henkilo(int p, int k, int v, std::string const& nimi,
2                   std::string const& hetu)
3  try // Huomaa try-sanan paikka!
4    : syntymapvm_(p, k, v), nimi_(nimi), hetup_(0)
5  {
          ⋮
15 }
16 catch (...)
17 { // Tänne päästään, jos jäsenmuuttujien (tai kantaluokan)
18   // luominen epäonnistuu. Tehdään tarvittaessa toimenpiteitä.
19   throw; // Tai heitetään jokin toinen poikkeus
20 }
```

LISTAUS 11.12: Funktion valvontalohko rakentajassa

jä ei tee näistä kumpaakaan, heitetään alkuperäinen virhe automaattisesti uudelleen.

11.8.3 Poikkeukset ja purkajat

Kuten aliluvussa 11.5 todettiin, poikkeuksen sattuessa kutsutaan automaattisesti kaikkien sellaisten olioiden purkajia, joiden elinkaari loppuu virhekäsittelijän etsimisen yhteydessä. Näin olion purkajaa saatetaan kutsua sellaisessa tilanteessa, jossa vireillä on jo yksi poikkeustilanne. Jos purkaja vielä vuotaa ulos toisen poikkeuksen, ei C++:n poikkeusmekanismi pysty selviytymään tilanteesta, vaan ohjelman suoritus keskeytetään kutsumalla funktiota `terminate`.

Tämän vuoksi on erittäin tärkeää, että ***olioiden purkajista ei vuoda poikkeuksia ulos***. Yleensä tämä ei ole ongelma, koska suurin osa siivoustoimenpiteistä — kuten esimerkiksi muistin vapauttaminen — on luonteeltaan sellaisia, että ne eivät voi epäonnistua. Mikäli purkajissa joudutaan kuitenkin tekemään toimenpiteitä, jotka voivat epäonnistua, niissä mahdollisesti tapahtuvat poikkeukset tulisi ottaa kiinni ja käsitellä jo purkajassa itsessään.

Toinen mahdollisuus on kirjoittaa luokalle erillinen `siivoa`-jäsenfunktio, joka suorittaa kaikki sellaiset siivoustoimenpiteet, jotka voivat epäonnistua. Luokan käyttäjien tulee sitten kutsua tätä jäsenfunktiota aina ennen olion tuhoamista, jolloin mahdolliset virheet voidaan ottaa kiinni. Tällainen erillinen siivousfunktio on kuitenkin kömpelö, eikä se edelleenkään ratkaise kysymystä siitä, mitä pitäisi

tehdä, jos yhden virhetilanteen jo vireillä ollessa kutsutaan siivousfunktiota ja havaitaan toinen virhe. Mainittakoon, että Java-kielessä tällaiset erilliset siivousfunktiot ovat C++:aa tavallisempia, koska kielessä ei ole C++:n purkajaa vastaavaa mekanismia.

On huomattava, etteivät edellä olleet asiat tarkoita sitä, etteikö luokan purkajassa saisi sattua poikkeusta. Jos näin tapahtuu, täytyy purkajan vain itse kyetä sieppaamaan syntynyt poikkeus ja toipumaan siitä. Oleellista on, ettei poikkeus *vuoda purkajasta ulos.* Periaatteessa siis kaikki purkajat tulisi kirjoittaa niin, että niille voisi antaa poikkeusmääreen **`throw()`**. Se, kirjoitetaanko tuo poikkeusmääre todella näkyville purkajaan, on sitten makuasia.

Kun oliota tuhotaan, myös sen kaikkien jäsenmuuttujien ja kantaluokkaosien purkajia kutsutaan. *Periaatteessa* C++ antaa mahdollisuuden ottaa kiinni näissä osaolioiden purkajissa sattuvat poikkeukset itse "emo-olion" purkajassa. Tähän tarkoitukseen purkajaan voi kirjoittaa samanlaisen funktion valvontalohkon kuin rakentajaan aliluvussa 11.8.2 sivulla 396. Tällöin tämän valvontalohkon virhekäsittelijään siirrytään, jos itse purkajasta tai jonkin jäsenmuuttujan tai kantaluokkaosan purkajasta vuotaa poikkeus ulos. Tälle mekanismille *ei* kuitenkaan ole käytännössä mitään käyttöä, koska purkajista ei koskaan tulisi vuotaa poikkeuksia, joten normaali purkajan sisällä oleva virhekäsittely on käytännössä aina riittävä.

Samoin C++ tarjoaa funktion `uncaught_exception`, joka palauttaa arvon **`true`**, jos ohjelmassa on vireillä jokin poikkeus. Joskus tätä näkee käytettävän siihen, että purkaja vuotaa poikkeuksen vain, jos vireillä ei ole toista poikkeusta. Tämäkään tapa *ei* ole suotava. Se ei nimittäin vastaa siihen kysymykseen, mitä tehdään jos vireillä tosiaan *on* toinen virhetilanne. Purkajassa tapahtuneen toisen virheen huomiotta jättäminen tuskin on kovin turvallista, eikä ole kovin järkevää, että purkaja toimii muutenkaan kahdella eri tavalla riippuen siitä, onko muualla havaittu virheitä. Yleinen mielipide onkin nykyisin, että `uncaught_exception`-funktio on C++:ssa käyttökelvoton, eikä funktion valvontalohkojakaan pitäisi käyttää muualla kuin korkeintaan rakentajissa. [Sutter, 2002c]

11.9 Esimerkki: poikkeusturvallinen sijoitus

Esimerkkinä poikkeusturvallisuuden huomioon ottamisesta käsitellään seuraavaksi listauksessa 11.13 esitetyn luokan `Kirja` sijoitusoperaattorin kirjoittamista. Listaus näyttää myös yhden mahdollisen sijoitusoperaattorin toteutuksen. Tämä sijoitusoperaattori on kirjoitettu aliluvun 7.2 ohjeiden mukaan, joten enää täytyy miettiä sen poikkeusturvallisuutta.

11.9.1 Ensimmäinen versio

Ensimmäinen vaihe on miettiä, millaisia virhemahdollisuuksia sijoituksessa voi sattua ja mitä niistä seuraa. Viitteiden ja osoittimien käsittelyissä ei virheitä voi sattua (ne antavat nothrow-takuun). Sen sijaan merkkijonojen sijoituksessa *voi* sattua virhe. Näin jäljelle jäävät seuraavat mahdolliset tapahtumasarjat:

1. Mitään virheitä ei satu, sijoitus saadaan suoritettua onnistuneesti. Tämä on tietysti toivottava lopputulos.

```
class Kirja
{
public:
      ⋮
  Kirja& operator =(Kirja const& kirja);
private:
  std::string nimi_;
  std::string tekija_;
};
```

```
Kirja& Kirja::operator =(Kirja const& kirja)
{
  if (this != &kirja)
  {
    nimi_ = kirja.nimi_;
    tekija_ = kirja.tekija_;
  }
  return *this;
}
```

LISTAUS 11.13: Yksinkertainen luokka, jolla on sijoitusoperaattori

2. On mahdollista, että heti ensimmäistä merkkijonoa `nimi_` sijoitettaessa tulee virhe. Tällöin sijoitus keskeytyy ja poikkeus vuotaa ulos sijoitusoperaattorista, jolloin toista merkkijonoa ei ehditä käsitellä lainkaan.

3. Viimeinen mahdollisuus on, että ensimmäinen sijoitus onnistuu, mutta toisessa tapahtuu virhe ja poikkeus vuotaa ulos sijoituksesta.

Vaihtoehto 1 ei luonnollisesti vaadi miettimistä poikkeusturvallisuuden kannalta. Sen sijaan tilanne vaihtoehdossa 2 riippuu siitä, millaisen poikkeustakuun `string` antaa omalle sijoitukselleen. C++-standardista (ja esim. teoksesta "The C++ Standard Library" [Josuttis, 1999]) käy ilmi, että suurin osa C++:n vakiokirjaston luokista, `string` mukaanlukien, tarjoaa käyttäjälleen perustakuun. Tämä tarkoittaa siis sitä, että virheen jälkeen olio on jossain käyttökelpoisessa, muttei välttämättä ennustettavassa tilassa. Käytännössä tämä tarkoittaa, että virheen sattuessa `string` sisältää jonkin järkevän merkkijonoarvon, mutta meillä ei ole tietoa siitä, mikä se on. Olion voi kuitenkin esim. tuhota onnistuneesti tai siihen voi yrittää sijoittaa uuden arvon.

`Kirja`-luokan kannalta tämä tarkoittaa, että vaihtoehdon 2 jälkeen kirjaolio itse on myös jossain käyttökelpoisessa muttei ennustettavassa tilassa. Vaikka kirjan tekijä on säilynytkin ennallaan, ei kirjan nimestä voida sanoa mitään! Jos sen sijaan `string` olisi tarjonnut vahvan poikkeustakuun ja luvannut, että virheen sattuessa sen arvo säilyy muuttumattomana, olisi tilanne ollut toinen. Silloin voitaisiin olla varmoja, että vaihtoehdon 2 sattuessa myös kirjaolion sisältö on säilynyt ennallaan, koska epäonnistuneen merkkijonosijoituksen lisäksi ei ehditty tehdä mitään muuta.

Vaihtoehto 3 on hieman ongelmallisempi. Siinä kirjan nimen sijoittaminen onnistuu, mutta tekijän sijoituksessa tulee virhe. Koska `string` tarjoaa käyttäjälleen perustakuun, on tuloksena tilanne, jossa kirjan nimi on muutettu, mutta merkkijono `tekija_` on jossain käyttökelpoisessa mutta ei ennustettavassa tilassa. Kirjan sijoitus on siis osaksi tapahtunut, osaksi epäonnistunut.

Tässä tapauksessa tilannetta ei muuttuisi, vaikka `string` olisikin tarjonnut vahvan poikkeustakuun. Lopputuloksena olisi tällöin kirja, jonka nimi olisi muutettu mutta tekijä olisi alkuperäinen. Kirjan tila ei siis olisi alkuperäinen eikä myöskään haluttu lopullinen, jo-

ten poikkeusturvallisuuden kannalta tässäkin tapauksessa kirjan tila olisi “käyttökelpoinen muttei järkevä”.

Kun kaikki vaihtoehdot otetaan huomioon, saadaan tulokseksi, että luokan `Kirja` sijoitus voi tarjota käyttäjälleen perustakuun. Pahin mahdollinen tilanne on, että virheen sattuessa kirjan tila on tuntematon, mutta kirja on kyllä muuten käyttökelpoinen, ja sen voi esimerkiksi tuhota tai siihen voi yrittää sijoittaa uudelleen.

Kuten tästä esimerkistä näkyy, kaikkien mahdollisten virhetilanteiden analysoiminen on varsin mutkikasta jo näinkin yksinkertaisessa luokassa. Tilanne muuttuu tietysti helposti erittäin hankalaksi, kun luokan rakenne monimutkaistuu.

11.9.2 Tavoitteena vahva takuu

Kuten esimerkki näyttää, perustakuun tarjoaminen on yleensä kohtalaisen helppoa, jos käytössä on muita perustakuun tarjoavia operaatioita. Perustakuu ei kuitenkaan ole käyttäjän kannalta paras mahdollinen, koska operaation epäonnistuessa ollaan hukattu olion alkuperäinen tila, mikä tekee esimerkiksi virheestä toipumisen vaikeaksi. Tämän vuoksi onkin hyvä miettiä, miten `Kirja`-luokan sijoituksen saisi tarjoamaan vahvan poikkeustakuun — virheen sattuessa kirjan tila olisi sama kuin ennen koko sijoitusta.

Ensin kannattaa miettiä, auttaisiko tilannetta, jos `C++` tarjoaisi `string`-luokan, joka tarjoaisi vahvan poikkeustakuun sijoitukselle. Tällöin äskeisen listan vaihtoehdot 1 ja 2 olisivat vahvan takuun mukaisia. Nimen sijoituksen epäonnistuminen säilyttää nimen alkuperäisenä, joten joko kirjan sijoitus onnistuu tai sitten se epäonnistuu ja kirjan tila säilyy alkuperäisenä. Sen sijaan vaihtoehto 3 tuottaa edelleen ongelmia. Sen tapahtuessa kirjan nimi on saatu sijoitettua, mutta tekijän arvo jää ennalleen sijoituksen epäonnistuttua.

Kirjan tila tässä tapauksessa olisi “käyttökelpoinen mutta ei toivottu”, mikä ei poikkeusturvallisuuden kannalta eroa olennaisesti perustakuun lupauksesta “käyttökelpoinen muttei ennustettava”. Kummassakin tapauksessa kirja on menettänyt alkuperäisen arvonsa, mutta ei ole saanut haluttua uutta arvoa. Siis kirja voisi tällä sijoitusoperaattorilla tarjota vain perustakuun, vaikka `string` tarjoaisikin vahvan takuun.

Listaus 11.14 seuraavalla sivulla näyttää seuraavan version sijoitusoperaattorista. Siinä yritetään kiertää äskeinen ongelma ottamal-

la talteen kirjan alkuperäinen nimi ja tekijä. Jos jumpikumpi sijoitus epäonnistuu, palautetaan nimi ja tekijä ennalleen. Tässä tapauksessa virhemahdollisuuksia tulee heti kaksi lisää, koska uusien merkkijonomuuttujien luominen voi epäonnistua. Näissä tapauksissa kirjan tilaan ei kuitenkaan ole ehditty koskea, joten nämä virheet eivät ole ristiriidassa vahvan takuun kanssa.

Ikävä kyllä, tämä yritelmä ei toimi. Mikään ei nimittäin takaa, että virheen jälkeen *alkuperäisten arvojen palauttaminen onnistuu* ilman virheitä! Arvojen palauttaminen virhekäsittelijässä riveillä 14–15 on samanlainen merkkijonojen sijoitus kuin muutkin, ja se voi epäonnistua, jolloin alkuperäisiä arvoja ei saadakaan palautettua niin kuin piti.

Lopputuloksena on, että `Kirja`-luokan sijoitus voi edelleen tarjota vain perustakuun, koska virheen jälkeen on mahdollista, että kirjan tila on käyttökelpoinen muttei ennustettava. Tällainen tilanne on käytännössä väistämätön, jos operaatio on suoritettava useassa osassa, ja virhe voi syntyä niin, että vain jotkin osista saadaan suoritettua.

```
1  Kirja& Kirja::operator =(Kirja const& kirja)
2  {
3     if (this != &kirja)
4     {
5        std::string vanhanimi(nimi_);
6        std::string vanhatekija(tekija_);
7        try
8        {
9           nimi_ = kirja.nimi_;
10          tekija_ = kirja.tekija_;
11       }
12       catch (...)
13       {
14          nimi_ = vanhanimi;
15          tekija_ = vanhatekija;
16          throw;
17       }
18    }
19    return *this;
20 }
```

LISTAUS 11.14: Sijoitus, joka pyrkii tarjoamaan vahvan takuun (ei toimi)

11.9.3 Lisätään epäsuoruutta

Ohjelmoinnin vanha sananlasku sanoo, että lähes minkä tahansa ongelman voi ratkaista lisäämällä ohjelmaan epäsuoruutta (yleensä osoittimia). Tämä viisaus pätee tässäkin tapauksessa. Edellisen yrityksen ongelmaksi muodostui, että sijoituksen epäonnistuessa vanhoja arvoja ei saatu palautettua.

Yksi ratkaisukeino on havaita, että vaikka merkkijonon sijoittaminen voi epäonnistua, niin *merkkijono-osoittimen* sijoittaminen onnistuu aina. Kirjan nimi ja tekijä laitetaankin dynaamisesti luotuina osoittimien päähän. Tällöin sijoituksessa uudesta nimestä ja tekijästä voidaan ensin luoda dynaamisesti erilliset kopiot. Jos kopioinnissa ei tapahdu virheitä, voidaan vanha nimi ja tekijä korvata uusilla ja tuhota vanhat ilman, että virheitä voi enää sattua. Listaus 11.15 näyttää tällaisen luokan ja listaus 11.16 seuraavalla sivulla sen toteutuksen.

Tämä toteutus tarjoaa vihdoin käyttäjälleen vahvan poikkeustakuun. Kirjan sijoitus joko onnistuu tai sitten tapahtuu virhe ja kirja säilyy alkuperäisessä tilassa. Tästä on varsin helppo varmistua, koska koodissa kirjan varsinaiseen tilaan kosketaan *vasta, kun kaikki mahdolliset virhepaikat on jo ohitettu.* Tämä on mahdollista, koska osoittimien sijoitus ja merkkijonon tuhoaminen eivät voi aiheuttaa poikkeuksia, joten kirjan vanhan tilan korvaaminen uudella ei voi keskeytyä, vaan se saadaan aina suoritettua onnistuneesti loppuun saakka. Samasta syystä sijoitusoperaattorissa normaalisti välttämätön itseen sijoituksen testaus ei ole enää välttämätön, koska olion vanha arvo

```
1  class Kirja
2  {
3  public:
4    Kirja(std::string const& nimi, std::string const& tekija);
5    // Tarvitaan myös oma kopiorakentaja (dynaaminen muistinhallinta)!
6    ~Kirja();
         ⋮
7    Kirja& operator =(Kirja const& kirja);
8  private:
9    std::string* nimip_;
10   std::string* tekijap_;
11 };
```

LISTAUS 11.15: Kirjaluokka epäsuoruuksilla

```
1  Kirja::Kirja(std::string const& nimi, std::string const& tekija)
2    : nimip_(0), tekijap_(0)
3  {
4    try
5    {
6      nimip_ = new std::string(nimi);
7      tekijap_ = new std::string(tekija);
8    }
9    catch (...)
10   {
11     delete nimip_; nimip_ = 0;
12     delete tekijap_; tekijap_ = 0;
13     throw;
14   }
15 }
16
17 Kirja::~Kirja()
18 {
19   delete nimip_; nimip_ = 0;
20   delete tekijap_; tekijap_ = 0;
21 }
22
23 Kirja& Kirja::operator =(Kirja const& kirja)
24 {
25   if (this != &kirja) // Periaatteessa tarpeeton!
26   {
27     std::string* uusinimip = 0;
28     std::string* uusitekijap = 0;
29     try
30     {
31       uusinimip = new std::string(*kirja.nimip_);
32       uusitekijap = new std::string(*kirja.tekijap_);
33       // Jos päästiin tänne, ei virheitä tullut
34       delete nimip_; nimip_ = uusinimip; // Onnistuvat aina
35       delete tekijap_; tekijap_ = uusitekijap; // Samoin nämä
36     }
37     catch (...)
38     {
39       delete uusinimip; uusinimip = 0;
40       delete uusitekijap; uusitekijap = 0;
41       throw;
42     }
43   }
44   return *this;
45 }
```

LISTAUS 11.16: Uuden kirjaluokan rakentaja, purkaja ja sijoitus

tuhotaan vasta uuden arvon luomisen jälkeen. Tehokkuusmielessä itseen sijoittamisen testaus saattaa silti olla paikallaan.

Esimerkistä huomaa selvästi, että tässä versiossa vahva poikkeustakuu on tehnyt luokasta selvästi kömpelömmän ja vaikeammin hallittavan. Lisäksi jokaisen kirjaolion luominen vaatii kaksi uutta dynaamista olion luomista, mikä tuhlaa hieman muistia ja hidastaa ohjelmaa vähäisessä määrin. Jos merkkijonoja olisi useampi kuin kaksi, muuttuisi tilanne aina vain pahemmaksi. Ratkaisua kannattaa siis vielä jalostaa.

11.9.4 Tilan eriyttäminen (*"pimpl"*-idiomi)

Vahvan poikkeustakuun antava toteutus saadaan tyylikkäämmäksi ja tehokkaammaksi, jos olion tila (sen jäsenmuuttujat) ja itse olio (sen "identiteetti") erotetaan toisistaan. Helpoimmin tämä tapahtuu laittamalla jäsenmuuttujat omaan **`struct`**-tietorakenteeseensa, johon oliossa on sitten osoitin. Tällaisella järjestelyllä voidaan olion tila korvata toisella yksinkertaisesti sijoittamalla osoittimen päähän uusi tilatietorakenne. Tässäkin tapauksessa tilan sisältävä **`struct`** täytyy luoda dynaamisesti, mutta luomisia tapahtuu vain yksi jäsenmuuttujien määrästä riippumatta. Lisäksi dynaamista muistinhallintaa voidaan helpottaa korvaamalla osoittimet aliluvun 11.7 automaattiosoittimilla.

Listaukset 11.17 seuraavalla sivulla ja 11.18 seuraavalla sivulla näyttävät esimerkin erillisen tilarakenteen käytöstä. Siinä `Kirja`-luokan sisällä on määritelty erillinen **`struct`**-tietorakenne `Tila`. Luokan esittelyn yhteydessä tästä tietorakenteesta riittää vain ennakkoesittely (aliluku 4.4), koska luokan esittelyssä tarvitaan vain (automaatti)osoitin tietorakenteeseen. Itse tietorakenteen määrittely voi olla vasta luokan toteuttavassa kooditiedostossa. Tällä tavalla ratkaisu myös lisää luokan kapselointia, koska jäsenmuuttujat eivät enää näy otsikkotiedostossa. Tällaisesta erillisestä tilatietorakenteesta käytetään englanninkielisessä C++-kirjallisuudessa yleisesti nimitystä "*pimpl*" (**p**rivate **impl**ementation)[1] ja se on joskus varsin käyttökelpoinen muutenkin kuin poikkeuksien yhteydessä [Sutter, 2000, kohdat 26–30].

[1] Lisäksi *pimpl* on ah, niin puujalka-hauskasti lähellä sanaa "pimple" = finni.

```
1  class Kirja
2  {
3  public:
4    Kirja(std::string const& nimi, std::string const& tekija);
5    // Tarvitaan myös oma kopiorakentaja!
6    // Oma purkaja tarvitaan, jotta auto_ptr ennakkoesittelylle toimii
7    ~Kirja();
         ⋮
8    Kirja& operator =(Kirja const& kirja);
9  private:
10   struct Tila;
11   std::auto_ptr<Tila> tilap_;
12 };
```

LISTAUS 11.17: Kirja eriytetyllä tilalla ja automaattiosoittimella

```
1  struct Kirja::Tila
2  {
3    std::string nimi_;
4    std::string tekija_;
5    Tila(std::string const& nimi, std::string const& tekija)
6      : nimi_(nimi), tekija_(tekija) {}
7  };
8
9  Kirja::Kirja(std::string const& nimi, std::string const& tekija)
10   : tilap_(new Tila(nimi, tekija))
11 {
12 }
13
14 Kirja::~Kirja()
15 { // Automaattiosoitin tuhoaa tilan automaattisesti
16 }
17
18 Kirja& Kirja::operator =(Kirja const& kirja)
19 {
20   std::auto_ptr<Tila> uusitilap(new Tila(*kirja.tilap_));
21   tilap_ = uusitilap; // Ei voi epäonnistua ja tuhoaa vanhan tilan
22   return *this;
23 }
```

LISTAUS 11.18: Eriytetyn tilan rakentajat, purkaja ja sijoitus

Automaattiosoittimen ansiosta luokan rakentajat ja purkajat näyttävät jo paljon siistimmiltä kuin aiemmin. Samoin sijoitusoperaattorissa ei enää tarvitse reagoida virheisiin, koska automaattiosoitin pitää huolen dynaamisesti luotujen tilatietorakenteiden tuhoamisesta. Sijoituksesta on myös jätetty tilasyistä pois itseen sijoituksen testaus, koska se ei enää ole välttämätön. Tämä ratkaisu on jo varsin ylläpidettävä ja tehokkuudeltaankin todennäköisesti siedettävä.

11.9.5 Tilan vaihtaminen päikseen

Nyt kun vahvan poikkeustakuun antavalle sijoitukselle on löytynyt yleispätevä ratkaisu, voidaan vielä tutkia, eikö nimenomaan esimerkin merkkijonojen tapauksessa voitaisi päästä tehokkaampaa ratkaisuun. Vähän `string`-luokan rajapintaa tutkimalla tällainen löytyykin. Luokka nimittäin tarjoaa jäsenfunktion `swap`, joka vaihtaa kahden merkkijonon arvot keskenään *nothrow-poikkeustakuulla*. Sama operaatio löytyy myös kaikista STL:n säiliöistä. Vaihto-operaatio antaa mahdollisuuden pitää merkkijonot `Kirja`-luokan normaaleina jäsenmuuttujina, mutta silti saavuttaa vahva poikkeustakuu. Listaus 11.19 seuraavalla sivulla näyttää esimerkin tästä.

Listauksessa luokkaan on lisätty johdonmukaisuuden vuoksi oma jäsenfunktio `vaihda`, joka vaihtaa kahden kirjan tilat keskenään käyttämällä `string`-luokan `swap`-operaatiota. Koska `swap` onnistuu aina, ei myöskään `vaihda`-jäsenfunktiossa voi tapahtua virhettä. Tätä käytetään hyväksi sijoitusoperaattorissa. Siellä sijoitettavasta oliosta luodaan ensin *kopio* uuteen paikalliseen `Kirja`-olioon. Tämä olio edustaa sitä, mihin sijoituksella halutaan päästä. Jos kopion luominen epäonnistuu, ei alkuperäiselle oliolle ole tehty mitään ja vahva poikkeustakuu pätee. Jos kopiointi onnistuu, vaihdetaan sijoituksessa yksinkertaisesti kopion ja vanhan kirjaolion tilat keskenään. Näin sijoitus tulee tehtyä. Sijoitusoperaattorista palattaessa vanhan tilan sisältävä paikallinen muuttuja lopuksi tuhoutuu.

Erillinen `vaihda`-jäsenfunktio on kätevä, koska sitä käyttämällä myös `Kirja`-olioita jäsenmuuttujinaan pitävät luokat voivat tarjota sijoitukselle vahvan poikkeustakuun samaa mekanismia käyttämällä. Lisäksi ohjelmassa saattaa muulloinkin olla kätevää pystyä vaihtamaan kahden olion tilat keskenään ilman virhemahdollisuutta.

Vihoviimeisenä esimerkkinä listaus 11.20 sivulla 409 yhdistää keskenään tilan eriyttämisestä saatavan lisäkapseloinnin ja tilan vaih-

```
1  class Kirja
2  {
3  public:
       ⋮
4    Kirja& operator =(Kirja const& kirja);
5    void vaihda(Kirja& kirja) throw();
6  private:
7    std::string nimi_;
8    std::string tekija_;
9  };
```

```
1  void Kirja::vaihda(Kirja& kirja) throw()
2  {
3    nimi_.swap(kirja.nimi_); // Ei voi epäonnistua
4    tekija_.swap(kirja.tekija_); // Eikä tämäkään
5  }
6
7  Kirja& Kirja::operator =(Kirja const& kirja)
8  {
9    Kirja kirjakopio(kirja); // Kopio sijoitettavasta
10   vaihda(kirjakopio); // Vaihdetaan itsemme siihen, ei epäonnistu
11   return *this; // Vanha tila tuhoutuu kirjakopion myötä
12 }
```

LISTAUS 11.19: Kirja, jossa on nothrow-vaihto

tamisen. Se tarjoaa ehkä kaikkein tyylikkäimmän yleiskäyttöisen ratkaisun, jolla vahva poikkeustakuu saadaan toteutettua lähes luokassa kuin luokassa.

```
 1 class Kirja
 2 {
 3 public:
 4   Kirja(std::string const& nimi, std::string const& tekija);
 5   Kirja(Kirja const& kirja);
 6   // Oma purkaja tarvitaan, jotta auto_ptr ennakkoesittelylle toimii
 7   ~Kirja();
         ⋮
 8   Kirja& operator =(Kirja const& kirja);
 9   void vaihda(Kirja& kirja) throw();
10 private:
11   struct Tila;
12   std::auto_ptr<Tila> tilap_;
13 };
```

```
 1 struct Kirja::Tila
 2 {
 3   std::string nimi_;
 4   std::string tekija_;
 5   Tila(std::string const& nimi, std::string const& tekija)
 6     : nimi_(nimi), tekija_(tekija) {}
 7 };
 8
 9 Kirja::Kirja(std::string const& nimi, std::string const& tekija)
10   : tilap_(new Tila(nimi, tekija))
11 {
12 }
13
14 Kirja::Kirja(Kirja const& kirja)
15   : tilap_(new Tila(*kirja.tilap_)) // Tilan kopiorakentaja
16 {
17 }
18
19 Kirja::~Kirja()
20 { // Automaattiosoitin tuhoaa tilan automaattisesti
21 }
22
23 void Kirja::vaihda(Kirja& kirja) throw()
24 {
25   std::auto_ptr<Tila> p(tilap_);
26   tilap_ = kirja.tilap_;
27   kirja.tilap_ = p;
28 }
29
30 Kirja& Kirja::operator =(Kirja const& kirja)
31 {
32   Kirja kirjakopio(kirja); // Kopio sijoitettavasta
33   vaihda(kirjakopio); // Vaihdetaan itsemme siihen, ei epäonnistu
34   return *this; // Vanha tila tuhoutuu kirjakopion myötä
35 }
```

LISTAUS 11.20: Yhdistelmä tilan eriyttämisestä ja vaihdosta