Luku 11

Virhetilanteet ja poikkeukset

Although the source of the Operand Error has been identified, this in itself did not cause the mission to fail. The specification of the exception-handling mechanism also contributed to the failure. In the event of any kind of exception, the system specification stated that: the failure should be indicated on the databus, the failure context should be stored in an EEPROM memory (which was recovered and read out for Ariane 501), and finally, the SRI processor should be shut down.

– ARIANE 5 *Flight 501 Failure Report [Lions, 1996]*

Virhetilanteisiin varautuminen ja niihin reagoiminen on aina ollut yksi vaikeimpia ohjelmoinnin haasteita. Virhetilanteissa ohjelman täytyy yleensä toimia normaalista poikkeavalla tavalla, ja näiden uusien ohjelman suoritusreittien koodaaminen tekee ohjelmakoodista helposti sekavaa. Lisäksi useiden erilaisten virhetilanteiden viidakossa ohjelmoijalta jää helposti tekemättä tarvittavia siivoustoimenpiteitä, kuten muistin vapautusta. Jos vielä lisäksi vaaditaan, että ohjelman täytyy toipua virheistä eikä vain esimerkiksi lopettaa ohjel-

man suoritusta virheilmoitukseen, ohjelmakoodin täytyy pystyä peruuttamaan virhetilanteen vuoksi kesken jääneet operaatiot.

Virhetilanteiden käsittely on kokonaisuudessaan niin laaja aihe, ettei siitä tämän teoksen puitteissa voida kertoa kovinkaan paljon, eikä se edes kovin oleellisesti liity tämän teoksen aiheeseenkaan. C++ tarjoaa kuitenkin virheiden käsittelyyn erityisen mekanismin, **poikkeukset** (*exception*), joiden toiminta perustuu luokkahierarkioihin ja näin ollen sivuaa myös olio-ohjelmointia. Virhekäsittelystä ja C++:n poikkeuksista löytyy lisätietoa esim. kirjoista "More Effective C++" [Meyers, 1996], "Exceptional C++" [Sutter, 2000] ja "More Exceptional C++" [Sutter, 2002c].

11.1 Mikä virhe on?

Useimmat tietokoneen käyttäjät sanovat ohjelman toimivan väärin, jos siinä ei ole heidän tarvitsemaansa ominaisuutta tai ohjelma antaa käyttäjän mielestä vääriä vastauksia. Ohjelmiston tekijän kannalta syyt näihin väitteisiin voivat olla määrittelyssä (yritetään tehdä ohjelmalla jotain mihin sitä alunperinkään ei ole tarkoitettu), suunnittelussa (toteutukseen ei ole otettu mukaan kaikkia määrittelyssä olleita asioita tai niiden toteutus on suunniteltu virheelliseksi) tai ohjelmoinnissa (ohjelmointityössä on tapahtunut virhe).

Tietokoneohjelmat ovat mutkikkaita ja paraskaan ohjelmisto ei luultavasti pysty varautumaan kaikenlaisiin eri tasoilla oleviin virhetilanteisiin etukäteen — vähintäänkin tällaisen ohjelmiston toteutuskustannukset nousisivat sietämättömiksi. Ohjelmistoa on kuitenkin helppo pitää kilpailijoitaan laadukkaampana, jos siitä löytyy muita enemmän virhetilanteisiin varautuvia ominaisuuksia.

Ohjelmointityössä ei pysty vaikuttamaan määrittelyn ja suunnittelun aikaisiin virheisiin, ne paljastuvat ohjelmiston testauksessa tai huonoimmassa tapauksessa vasta tuotantokäytössä. Ohjelmoinnissa voidaan varautua etukäteen pohdittuihin vikatilanteisiin, jotka voidaan karkeasti jakaa laitteiston ja ohjelmiston aiheuttamiin.

Laitteistovirheet näkyvät ohjelmistolle sen ympäristön käyttäytymisenä eri tavoin kuin on oletettu. Ohjelmia tehtäessä oletetaan esimerkiksi, että muuttujaan kirjoitettu arvo on säilynyt samana, kun sitä hetken kuluttua luetaan muuttujasta — viallinen muistipiiri tietokoneessa saattaa kuitenkin aiheuttaa tilanteen olevan toinen. Tie-

dostojen käsittely voi mennä vikaan levyn täyttymisen tai vikaantumisen vuoksi. Ohjelma itsessään on saattanut osittain muuttua kun se on ladattu suoritettavaksi ja näin ollen toimii väärin suorittaessaan konekäskyjä, joita ohjelmoija ei ole tarkoittanut suoritettavaksi.

Laitteistovirheistä saadaan tietoa yleensä käyttöjärjestelmän kautta. Tiedostojen käsittelyssä tapahtuneet virheet useimmat käyttöjärjestelmät osaavat ilmoittaa ohjelmalle, mutta muut “vakavammat” laitevirheet voivat aiheuttaa tiedon muuttumista ilman, että siitä erikseen tulee ilmoitusta ohjelmalle. Ohjelmoijan on todellisuudessa aina tehtävä jonkinlainen kompromissi sen kanssa, minkä tyyppisiä virheitä ohjelmassa pyritään havaitsemaan ja käsittelemään. Laitteiston tapauksessa yleisin linja on varautua käyttöjärjestelmän ilmoittamiin vikoihin ja jättää muut huomioimatta luottaen niiden olevan erittäin harvinaisia.

Erään arvion mukaan nykyaikainen henkilökohtainen tietokone tekee virheen laitteiston laskutoimituksissa keskimäärin kolmen triljoonan (18 nollaa) laskun suorituksen jälkeen. Tämä tarkoittaa suunnilleen sitä, että ajettaessa ohjelmistoa tällaisella koneella tuhat vuotta vika esiintyy kerran. Useimmat ohjelmistot jättävät nämä vikamahdollisuudet tarkastamatta, mutta joskus nekin muodostuvat merkittäviksi. Esimerkiksi massiivista rinnakkaiseksi hajautettua laskentaa suorittava *SETI@home*-projekti käyttää edellä mainitun ajan prosessoriaikaa päivässä ja törmää kyseiseen vikaan siis keskimäärin kerran vuorokaudessa — tällöin vikamahdollisuus on myös huomioitava ohjelmistossa. [SETI, 2001]

Ohjelmistossa virheet ovat yksittäisen ohjelmanpätkän (funktio, olio, moduuli) kannalta sisäisiä tai ulkoisia. Ulkoisessa virheessä koodia pyydetään tekemään jotain, mitä se ei osaa tai mihin se ei pysty. Esimerkiksi funktion parametrilla on väärä arvo, syötetiedosto ei noudata määriteltyä muotoa, tai käyttäjä on valinnut toimintosekvenssin jossa ei ole “järkeä”. Sisäisessä virheessä toteutus ajautuu itse tilanteeseen jossa jotain menee pieleen (esimerkiksi muisti loppuu tai toteutusalgoritmissa tulee jokin ääriraja vastaan).

Virhetilanteita huomioivassa ohjelmoinnissa on usein kaikista helpoin vaihe havaita virhetilanne. Tähän toimintaan käyttöjärjestelmät, ohjelmakirjastot ja ohjelmointikielet tarjoavat lähes aina keinoja. Havaitsemista paljon vaikeampaa on suunnitella ja toteuttaa se, mitä vikatilanteessa tehdään.

11.2 Mitä tehdä virhetilanteessa?

Varautuva ohjelmointi (*defensive programming*, [McConnell, 1993]) on ohjelmointityyli, jota voisi verrata autolla ajossa ennakoivaan ajotapaan. Vaikka oma toiminta (koodi) olisi täysin oikein ja sovittujen sääntöjen mukaista, kannattaa silti varautua siihen, että muut osallistujat voivat toimia väärin. Usein ajoissa tapahtunut virheiden ja ongelmien havaitseminen mahdollistaa niihin sopeutumisen jopa siten, että ohjelmissa käyttäjän ei tarvitse huomata mitään erityistilannetta edes syntyneen.

Seuraavassa on listattu muutamia tapoja, joilla ohjelman osa voi toimia havaitessaan virhetilanteen. Sopiva suhtautuminen virheeseen on vähintäänkin ohjelmakomponentin suunnitteluun kuuluva asia. Ei ole olemassa yhtä ainoata oikeata tai väärää tapaa — hyvin suunniteltu komponentti voi ottaa virheisiin reagoinnin omalle vastuulleen, mutta hyvänä ratkaisuna voidaan pitää myös sellaista, joka "ainoastaan" ilmoittaa havaitsemansa virheet komponentin käyttäjälle.

- Suorituksen keskeytys (abrupt termination) on äärimmäinen tapa toimia kun ohjelmassa kohdataan virhe. Järjestelmän suorittaminen keskeytetään välittömästi ja usein ilman, että virhetilannetta yritetään edes mitenkään kirjata myöhempää tarkastelua varten. Tämän tavan käyttöä tulisi välttää, sillä pysähtyneestä ohjelmasta ei edes aina tiedetä miksi pysähtyminen tapahtui. Valitettavan useassa käyttöjärjestelmässä ja ohjelmointikielten ajoympäristöissä tämä on oletustoiminta silloin kun jokin virhe on havaittu (esimerkiksi kaatuminen muistin loppuessa).

- Suorituksen hallittu lopetus (abort, exit)[ϒ] on edellistä lievempi tapa, jossa yritetään siivota ohjelmiston tila vapauttamalla kaikki sen varaamat resurssit ja kirjaamalla virhetilanne pysyvään talletuspaikkaan sekä ilmoittamalla virheestä käyttäjälle ennen suorituksen lopettamista.

- Jatkaminen (continuation) tarkoittaa havaitun virheen jättämistä huomiotta. Määrittelynsä mukaisesti virhe on ohjelman ei-

[ϒ]Huom: C++-kielen funktiot `abort()` ja `exit()` toteuttavat suorituksen keskeytyksen, eivät hallittua lopetusta.

toivottu tila, joten sellaisen jättäminen käsittelemättä havainnoinnin jälkeen on hyvin hyvin harvinainen toimintamalli. Joskus esimerkiksi käyttöliittymässä tapahtumien puuttuminen tai katoaminen voi olla tilanne, jossa jatkaminen tulee kysymykseen — esimerkiksi yksittäiset hiirikohdistimen paikkatietoa sisältävät tapahtumat voivat kadota ilman että tilanteella on mitään vaikutusta ohjelmiston toimintaan.

- Peruuttaminen (rollback) tarkoittaa järjestelmän tilan palauttamista siihen tilanteeseen, mikä se oli ennen virheen aiheuttaneen operaation käynnistämistä. Tämä helpottaa huomattavasti esimerkiksi operaation yrittämistä uudelleen, koska tiedetään tarkasti missä tilassa ohjelmisto on, vaikka virhe onkin tapahtunut. Valitettavasti peruuttamisen toteuttaminen on usein mutkikasta (voi aiheuttaa itsessään virheitä ohjelmistoon) ja resursseja kuluttavaa. Yksi yksinkertainen toteutustapa on operaation alussa luoda kopio muutettavasta datasta ja tehdä muutokset tähän kopioon. Jos operaatio menee läpi ilman virheitä, vaihdetaan kopio alkuperäisen datan tilalle ja virheiden sattuessa tuhotaan kopioitu data (alkuperäinen pysyy koskemattomana).

- Toipuminen (recovery) on ohjelman osan paikallinen toteutus hallitusta lopetuksesta. Osanen ei pysty itse käsittelemään havaittua virhettä, mutta se pyrkii vapauttamaan kaikki varaamansa resurssit ennen kuin virheestä tiedotetaan ohjelmistossa toisaalle (usein loogisesti ylemmälle tasolle) jonka toivotaan pystyvän käsittelemään havaittu virhe paremmin.

Peruuttaminen ja toipuminen antavat mahdollisuuden yrittää korjata tilannetta, joka johti virhetilanteeseen. Pieleen menneen operaation yrittäminen uudelleen on virheisiin reagoinnin suunnittelussa hankalinta. Helpoimmassa tapauksessa operaatio vain toistetaan (esimerkiksi tietoliikenteessä suoritetaan uudelleenlähetys), mutta valitettavan usein virhetilanne johtuu ongelmista resursseissa, joiden puuttuessa tilanteen korjaaminen on hankalaa.

Yksi hyvä esimerkki on muistin loppuminen. Ohjelmointikielet ja -ympäristöt tarjoavat lähes aina tavan havaita, että ohjelman suorituksen aikana siltä on loppunut muisti (viimeisin dynaamisen muistin varausoperaatio on epäonnistunut). Tilanteen voi havaita, mutta

mitä sille voi tehdä? Jos muistia ei ole, niin toipumisoperaatiot eivät missään tapauksessa saa kuluttaa lisää muistia.

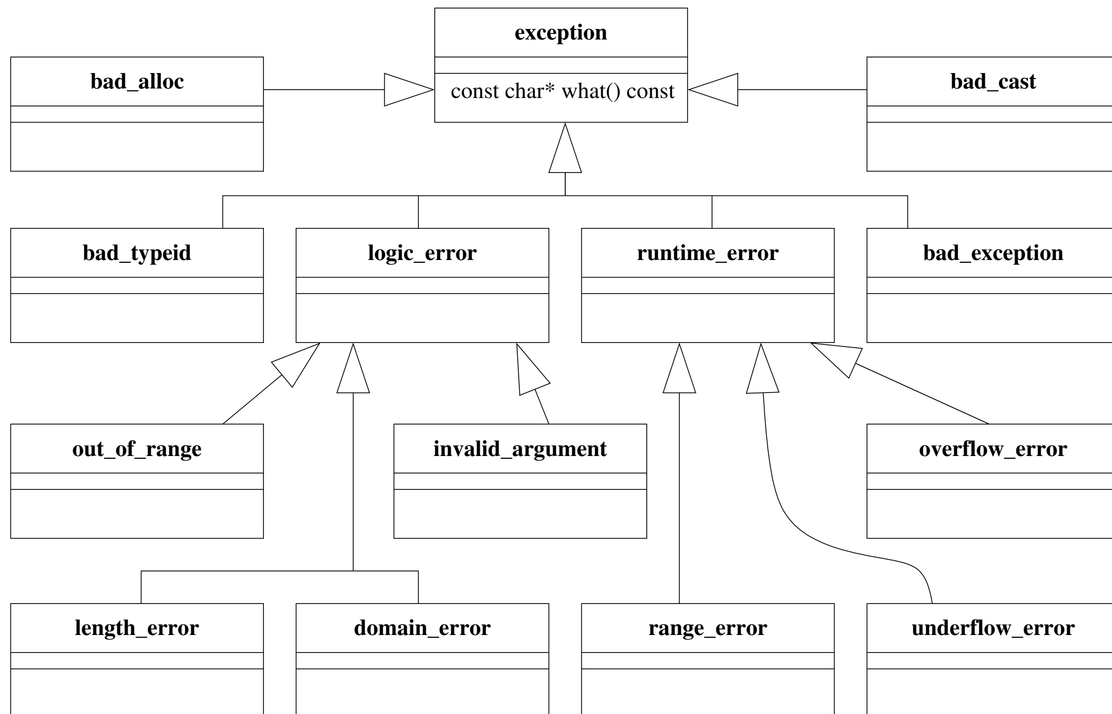
Järjestelmästä voitaisiin yrittää vapauttaa käytössä olevaa muistia, mutta resursseista järkevästi huolta pitävä ohjelmisto on tietysti alunperin toteutettu siten, ettei turhia muistivarauksia ole olemassa. Seuraavaksi voidaan etsiä "vähemmän tärkeitä" muistivarauksia ja vapautetaan ne, jolloin virhetilanne hyvin suurella todennäköisyydellä siirtyy toisaalle ohjelmistossa.

Yksi varma tapa saada muistia lisää käyttöönsä on varautua loppumiseen etukäteen varaamalla kasa "turhaa" muistia, joka voidaan turvallisesti vapauttaa uusiokäyttöön, jos joudutaan tilanteeseen, jossa muisti on lopussa [Meyers, 1996]. Muistia jatkuvasti syövän virheen tapauksessa tietysti vain pitkitetään todellisen ongelman kohtaamista, mutta ohimeneviin muistiongelmiin tämä on täyden toipumisen toteuttava tapa.

11.3 Virhehierarkiat

Ohjelmassa tapahtuvat mahdolliset virhetilanteet voidaan usein jakaa kategorioihin sen perusteella, mihin virhe liittyy. Tällaista virheiden jaottelua voi kuvata luokkakaaviolla niin kuin tavallistenkin olioiden kategorioita. Kuva 11.1 seuraavalla sivulla näyttää esimerkinä, miten tämä jako on tehty C++:n standardikirjastossa. Kuvan hierarkia ei tietenkään ole täydellinen, mutta se kattaa C++:n itsensä tuottamat poikkeukset. Ohjelmoija voi itse laajentaa tätä hierarkiaa tai kirjoittaa oman hierarkiansa alusta saakka, jos niin haluaa.

Tällaisten virhehierarkioiden etuna on, että ne tekevät mahdolliseksi virhekäsittelyn jakamisen eri tasoihin. Esimerkiksi kuvan hierarkiassa virheet jakautuvat kahteen pääkategoriaan: logiikkavirheet (`logic_error`) ja ajoaikavirheet (`runtime_error`). Logiikkavirheisiin kuuluvat kaikki sellaiset virheet, jotka aiheutuvat ohjelman toimintalogiikassa havaituista virheistä, "bugeista". Tällaisen virheen tapahtuminen on merkki siitä, että ohjelmassa on vikaa. Ajoaikavirheet puolestaan johtuvat siitä, että ohjelman suorituksen aikana ajoympäristö aiheuttaa tilanteen, jota ohjelma ei pysty hallitsemaan. Esimerkkejä tästä ovat ylivuodot (ohjelmalle syötetään liian suuria lukuja), virheet lukualueissa (käyttäjä syöttää kuukauden numeroksi 13) ja vaikkapa tietoliikenneyhteyden katkeaminen.



KUVA 11.1: C++-standardin virhekategoriat

Kun virheet on jaoteltu hierarkiaksi, jotkin ohjelman osat voivat esimerkiksi käsitellä ylivuodot ja kenties toipua niistä, mutta jättävät muut virheet ohjelman ylempien tasojen huoleksi. Ylempi ohjelman osa voi sitten käsitellä yhtenäisesti kaikkia ajoaikaisia virheitä välittämättä siitä, mikä nimenomainen virhe on kyseessä.

Koska virheiden muodostama hierarkia muistuttaa suuresti luokkahierarkiaa, olio-ohjelmoinnissa voidaan virheitä mallintaa luokilla, jotka toteutetaan ohjelmassa. Näitä luokkia voidaan sitten käyttää hyväksi C++:n poikkeusten kanssa. Listaus 11.1 seuraavalla sivulla näyttää osan kuvan 11.1 virhetyypeistä luokkina, jotka löytyvät C++:n standardikirjastoista <`exception`> ja <`stdexcept`>.

Periaatteessa ohjelmoija voi itse vapaasti päättää, käyttääkö itse alusta saakka suunnittelemaansa virhehierarkiaa vai periyttääkö tarvitsemansa poikkeukset kirjaston virhehierarkiasta. C++:n oman virhehierarkian laajentaminen tietysti yhtenäistää virheiden käsittelyä, joten se lienee usein tarkoituksenmukaista. Hierarkiaan voi tietysti

```
 1 // Nämä kaikki ovat std-nimiavaruudessa
 2 class exception
 3 {
 4 public:
 5    exception() throw(); // throw() selitetään myöhemmin
 6    exception(exception const& e) throw();
 7    exception& operator =(exception const& e) throw();
 8    virtual ~exception() throw();
 9    virtual char const* what() const throw();
          ⋮
10 };
11
12 class runtime_error : public exception
13 {
14 public:
15    runtime_error(std::string const& msg);
16 };
17
18 class overflow_error : public runtime_error
19 {
20 public:
21    overflow_error(std::string const& msg);
22 };
```

LISTAUS 11.1: Virhetyypit C++:n luokkina

lisätä myös uusia alihierarkioita periyttämällä ne standardin kantaluokista.

Listaus 11.2 seuraavalla sivulla näyttää esimerkin omasta virheluokasta `LiianPieniArvo`, joka kuvaa virhettä, jossa jokin ohjelman arvo on liian pieni sallittuun verrattuna. Tämä on erikoistapaus C++:n virhetyypistä `domain_error`, joten oma virheluokka on periytetty siitä. Listauksesta näkyy myös, kuinka virheluokkaan voi upottaa tietoa itse virheestä. Tässä tapauksessa jokainen `LiianPieniArvo`-olio sisältää tiedon siitä, mikä liian pieni arvo oli ja mikä arvon minimiarvo olisi ollut. Nämä tiedot annetaan oliolle rakentajan parametrina, kun virhetilanne havaitaan ja virheolio luodaan. Virhettä käsitellessä arvoja voi sitten kysyä luokan anna-jäsenfunktioilla.

```
1  class LiianPieniArvo : public std::domain_error
2  {
3  public:
4     LiianPieniArvo(std::string const& viesti, int luku, int minimi);
5     LiianPieniArvo(LiianPieniArvo const& virhe);
6     virtual ~LiianPieniArvo() throw();
7     int annaLuku() const;
8     int annaMinimi() const;
9  private:
10    int luku_;
11    int minimi_;
12 };
```

LISTAUS 11.2: Esimerkki omasta virheluokasta

11.4 Poikkeusten heittäminen ja sieppaaminen

C++:n poikkeusten periaatteena on, että virheen sattuessa ohjelma **heittää** (*throw*) "ilmaan" poikkeusolion, joka kuvaa kyseistä virhettä. Tämän jälkeen ohjelma alkaa "peruuttaa" funktioiden kutsuhierarkiassa ylöspäin ja yrittää etsiä lähimmän **poikkeuskäsittelijän** (*exception handler*), joka pystyy **sieppaamaan** (*catch*) virheolion ja reagoimaan virheeseen. Jokaisella poikkeuskäsittelijällä on oma koodilohkonsa, **valvontalohko** (*try-block*), jonka sisällä syntyvät virheet ovat sen vastuulla. *Virhekäsittelyn yhteydessä poikkeusoliosta tehdään kopio, joten on tärkeää, että poikkeusluokilla on toimiva kopiorakentaja.*

Poikkeuksen heittäminen ja poikkeuskäsittelijän etsiminen on kohtalaisen raskas operaatio verrattuna useimpiin muihin C++:n operaatioihin. Niinpä onkin tärkeää, että poikkeuksia käytetään vain *poikkeuksellisten* tilanteiden käsittelyyn eikä esimerkiksi uutena muodikkaana hyppykäskynä.

11.4.1 Poikkeushierarkian hyväksikäyttö

Listaus 11.3 sivulla 376 sisältää esimerkin virhekäsittelystä keskiarvon laskennassa. Keskiarvoa laskettaessa on kaksi virhemahdollisuutta: lukujen lukumäärä saattaa olla nolla tai niiden summa saattaa kasvaa liian suureksi. Lukujen summaa laskeva funktio heittää ylivuoto-

tapauksessa riveillä 10 ja 15 poikkeuksen komennolla **`throw`**.[♉] Vastaavasti keskiarvon laskeva funktio heittää poikkeuksen rivillä 27, jos lukujen lukumäärä on nolla.

Funktio `keskiarvo1` sisältää kaksi poikkeuskäsittelijää riveillä 40–47. Käsittelijä merkitään avainsanalla **`catch`**, jonka jälkeen annetaan parametrilistan omaisesti käsittelijän hyväksymän poikkeusolion tyyppi. Poikkeuskäsittelijän parametrit on syytä merkitä vakioviitteiksi samasta syystä kuin tavallisetkin parametrit olioita välitettäessä (aliluku 4.3.3). Poikkeuskäsittelijät pystyvät sieppaamaan virhetilanteet, jotka syntyvät sinä aikana, kun ohjelman suoritus on riveillä 34–39 olevan **`try`**-avainsanalla merkityn valvontalohkon sisällä. Ensimmäinen poikkeuskäsittelijä sieppaa nollallajakovirheet, jälkimmäinen ylivuodot.

Normaalissa tapauksessa ohjelman suoritus siirtyy valvontalohkoon, suorittaa siellä koodin ja hyppää sen jälkeen poikkeuskäsittelijöiden yli riville 48. Tällä tavoin virhekäsittely ei millään tavalla vaikuta ohjelman normaaliin toimintaan.

Jos lukujen summa kasvaa liian suureksi, rivi 10 heittää poikkeuksen. Ohjelma alkaa tällöin etsiä lähintä sopivaa poikkeuskäsittelijää. Funktiossa `summaaLuvut` sellaista ei ole, joten virhe “vuotaa” ulos tästä funktiosta ja ohjelma peruuttaa takaisin funktioon `laskeKeskiarvo`. Sielläkään ei ole poikkeuskäsittelijää, joten ohjelma palaa funktioon `keskiarvo1`. Siellä on vihdoin ylivuotovirheen hyväksyvä poikkeuskäsittelijä, ja ohjelman suoritus jatkuu poikkeuskäsittelijän koodista riviltä 46.

Virhekäsittelyn päätyttyä ohjelma ***ei palaa virhekohtaan*** vaan jatkuu koko virhekäsittelyrakenteen jälkeen riviltä 48. Virhekäsittelyä ei siis suoraan voi käyttää täydelliseen virheestä toipumiseen, jossa ohjelman suoritus palaisi virhekäsittelyn jälkeen takaisin valvontalohkoon jatkamaan sen suoritusta.

Listauksen 11.3 molemmat poikkeuskäsittelijät sisältävät lähes saman koodin, koska molemmat virheet ovat luonteeltaan samanlaisia. Joskus onkin järkevää tehdä poikkeuskäsittelijä, joka sieppaa kaikki *tiettyyn virhekategoriaan* kuuluvat poikkeukset. Tämä tehdään laittamalla poikkeuskäsittelijän parametriksi viite virhehierarkian haluttuun kantaluokkaan. Koska jokainen virhekantaluokasta peritty virhe on olio-ohjelmoinnin mukaan myös kantaluokan olio, poikkeus-

[♉] Koodissa `numeric_limits<double>::max()` palauttaa suurimman mahdollisen liukuluvun. `numeric_limits`-mallia käsiteltiin aliluvussa 10.6.4.

```
1  void lueLuvutTaulukkoon(vector<double>& taulu);
2
3  double summaaLuvut(vector<double> const& luvut)
4  {
5    double summa = 0.0;
6    for (unsigned int i = 0; i < luvut.size(); ++i)
7    {
8      if (summa >= 0 && luvut[i] > numeric_limits<double>::max()-summa)
9      {
10       throw std::overflow_error("Summa liian suuri");
11     }
12     else if (summa < 0 &&
13              luvut[i] < -numeric_limits<double>::max()-summa)
14     {
15       throw std::overflow_error("Summa liian pieni");
16     }
17     summa += luvut[i];
18   }
19   return summa;
20 }
21
22 double laskeKeskiarvo(vector<double> const& luvut)
23 {
24   unsigned int lukumaara = luvut.size();
25   if (lukumaara == 0)
26   {
27     throw std::range_error("Lukumäärä keskiarvossa 0");
28   }
29   return summaaLuvut(luvut) / static_cast<double>(lukumaara);
30 }
31
32 void keskiarvo1(vector<double>& lukutaulu)
33 {
34   try
35   {
36     lueLuvutTaulukkoon(lukutaulu);
37     double keskiarvo = laskeKeskiarvo(lukutaulu);
38     cout << "Keskiarvo: " << keskiarvo << endl;
39   }
40   catch (std::range_error const& virhe)
41   {
42     cerr << "Lukualuevirhe: " << virhe.what() << endl;
43   }
44   catch (std::overflow_error const& virhe)
45   {
46     cerr << "Ylivuoto: " << virhe.what() << endl;
47   }
48   cout << "Loppu" << endl;
49 }
```

LISTAUS 11.3: Esimerkki C++:n poikkeuskäsittelijästä

käsittelijä sieppaa myös kaikki kantaluokasta periytetyt poikkeukset. Listauksessa 11.4 on uusi keskiarvofunktio, jonka poikkeuskäsittelijä sieppaa kaikki ajoaikaiset virheet.

11.4.2 Poikkeukset, joita ei oteta kiinni

Ohjelmassa voi tietysti tapahtua myös poikkeus, jota mikään poikkeuskäsittelijä ei sieppaa. Esimerkissä on mahdollista, että lukuja taulukkoon luettaessa muisti loppuu. Tällöin `vector`-luokka vuotaa ulos poikkeuksen `std::bad_alloc` (katso aliluku 3.5.1). Jos nyt funktio `lueLuvutTaulukkoon` ei itse sieppaa tätä virhettä ja toivu siitä, vuotaa virhe ulos tästäkin funktiosta.

Mikäli virhe pääsee vuotamaan ulos pääohjelmastakin eli jos ohjelmassa ei yksinkertaisesti ole sopivaa poikkeuskäsittelijää, ohjelma kutsuu funktiota `terminate`. Oletusarvoisesti tämä funktio vain lopettaa ohjelman suorituksen (ja kenties tulostaa virheilmoituksen). Ohjelma voi itse tarjota oman toteutuksensa tälle funktiolle, mutta joka tapauksessa ohjelman suoritus loppuu tämän funktion suoritukseen.

Koska poikkeukset, joihin ei ole varauduttu, aiheuttavat ohjelman kaatumisen, on tärkeää, että ohjelmassa otetaan jollain tasolla kiinni kaikki aiheutetut poikkeukset. Joissain tapauksissa voi tietysti olla, että ohjelman kaatuminen on hyväksyttävä reaktio virhetilanteeseen.

```
1  void keskiarvo2(vector<double>& lukutaulu)
2  {
3     try
4     {
5        lueLuvutTaulukkoon(lukutaulu);
6        double keskiarvo = laskeKeskiarvo(lukutaulu);
7        cout << "Keskiarvo: " << keskiarvo << endl;
8     }
9     catch (std::runtime_error const& virhe)
10    {
11       // Tänne tullaan minkä tahansa ajoaikaisen virheen seurauksena
12       cerr << "Ajoaikainen virhe: " << virhe.what() << endl;
13    }
14
15    cout << "Loppu" << endl;
16 }
```

LISTAUS 11.4: Virhekategorioiden käyttö poikkeuksissa

Ohjelmaan voi myös lisätä “yleispoikkeuskäsittelijöitä”, jotka ottavat vastaan *kaikki* valvontalohkossaan tapahtuvat poikkeukset. Yleispoikkeuskäsittelijän syntaksi on

```
catch (...) // Todellakin ... eli kolme pistettä
{
  // Tämä poikkeuskäsittelijä sieppaa kaikki poikkeukset
}
```

Yleensä tällaisia “kaikkivoipia” yleispoikkeuskäsittelijöitä ei kannata kirjoittaa kuin korkeintaan pääohjelmaan, ellei sitten ole aivan varma, että poikkeuskäsittelijän koodi todella pystyy reagoimaan oikein *kaikkiin mahdollisiin* poikkeuksiin, joita valvontalohkossa voi sattua. Yleispoikkeuskäsittelijähän sieppaa myös sellaiset virheet, joihin kenties voitaisiin paremmin reagoida ylemmällä tasolla ohjelmassa!

Poikkeuksen (☺) tästä muodostavat tilanteet, joissa virheestä riippumatta täytyy suorittaa tietty siivouskoodi, jonka jälkeen *virhe heitetään uudelleen* komennolla **throw;** ylemmän tason poikkeuskäsittelijöiden hoidettavaksi. Tällöin yleispoikkeuskäsittelijä on varsin käyttökelpoinen. Poikkeuksista ja siivoustoimenpiteistä kerrotaan enemmän aliluvussa 11.5.

11.4.3 Sisäkkäiset valvontalohkot

Listauksien 11.3 ja 11.4 keskiarvofunktiot eivät ota mitään kantaa muistin loppumiseen, joten niissä mahdollisesti syntyvät muistin loppumisesta aiheutuvat poikkeukset — tai mitkä tahansa poikkeukset ajoaikavirheitä lukuun ottamatta — vuotavat funktioista ulos ylempiin funktioihin. Niissä voi puolestaan olla omia poikkeuskäsittelijöitään, joista jotkin voivat sitten siepata muistin loppumisesta aiheutuneet poikkeukset. Jos poikkeus heitetään usean sisäkkäisen valvontalohkon sisällä, etsitään “lähin” poikkeuskäsittelijä, joka pystyy sieppaamaan poikkeuksen.

Listaus 11.5 seuraavalla sivulla näyttää pääohjelman, joka kutsuu keskiarvofunktiota ja sisältää lisäksi oman poikkeuskäsittelijänsä. Keskiarvofunktio käsittelee itse ajoaikavirheet, mutta muut virheet vuotavat keskiarvofunktiosta ulos pääohjelmaan. Näistä pääohjelma käsittelee itse muistin loppumisen ja kaikki ohjelman `Virhe`-luokasta periytetyt poikkeukset.

```
 1 int main()
 2 {
 3   vector<double> taulu;
 4   try
 5   {
 6     keskiarvo2(taulu); // Lue luvut ja laske keskiarvo
 7   }
 8   catch (std::bad_alloc&)
 9   {
10     cerr << "Muisti loppui!" << endl;
11     return EXIT_FAILURE;
12   }
13   catch (std::exception const& virhe)
14   {
15     cerr << "Virhe pääohjelmassa: " << virhe.what() << endl;
16     return EXIT_FAILURE;
17   }
18
19   return EXIT_SUCCESS;
20 }
```

LISTAUS 11.5: Sisäkkäiset valvontalohkot

Tämä mahdollisuus *sisäkkäisiin* valvontalohkoihin on erittäin hyödyllinen ominaisuus, varsinkin kun se yhdistetään virheluokkahierarkioihin. Tällöin ohjelman alemmissa osissa voidaan käsitellä yksityiskohtaisesti tietyt virheet, kuten esimerkissä keskiarvosta johtuvat ylivuodot ja nollalla jakaminen. Ohjelman ylemmät osat taas voivat käsitellä yleisemmällä tasolla laajempia virhekategorioita. Näin jokainen ohjelman osa voi reagoida virheisiin omalla abstraktiotasollaan. Triviaalit pikkuvirheet siepataan alemmilla tasoilla ja ylätasoille vuotavat suuremmat virheet voivat puolestaan aiheuttaa dramaattisempia toimia.

Poikkeuskäsittelijä voi myös halutessaan heittää virheen edelleen, jos se ei pysty toipumaan virheestä. Tämä saadaan aikaan poikkeuskäsittelijän koodissa komennolla **`throw;`** ilman mitään parametria. Tällaista osittaista poikkeuskäsittelyä käytetään hyväksi funktion siivoustoimenpiteissä seuraavassa aliluvussa. Poikkeuskäsittelijä voi myös muuttaa poikkeuksen toiseksi heittämällä omasta koodistaan uuden poikkeuksen.

11.5 Poikkeukset ja olioiden tuhoaminen

Poikkeuksen sattuessa ohjelma palaa takaisin koodilohkoista ja funktioista, kunnes se löytää sopivan poikkeuskäsittelijän. Tämän peruuttamisen tuloksena saatetaan poistua usean olion ja muuttujan näkyvyysalueelta. C++ pitää huolen siitä, että *kaikki oliot ja muuttujat, joiden näkyvyysalue loppuu poikkeuksen tuloksena, tuhotaan normaalisti*. Olioiden purkajia kutsutaan, joten niiden siivoustoimenpiteet suoritetaan kuten pitääkin. Näin poikkeukset eivät aiheuta mitään ongelmia olioille, joiden elinkaari on staattisesti määrätty.

Java-kielessä ei ole purkajia samalla tavoin kuin C++:ssa, joten siinä kielen muuten C++:aa muistuttavaan poikkeuskäsittelyyn on lisätty erityinen poikkeuskäsittelijöiden jälkeen tuleva **`finally`**-lohko, jossa oleva koodi suoritetaan aina lopuksi, tapahtui valvontalohkossa poikkeus tai ei. Tähän lohkoon voi kirjoittaa siivoustoimenpiteitä, jotka suoritetaan aina valvontalohkosta poistumisen jälkeen.

Joskus vastaavasta siivouslohkosta olisi hyötyä myös C++-kielessä, mutta sen poikkeusmekanismista ei tällaista löydy. Yleisesti käytetty ratkaisu on upottaa mahdollisimman moni siivousta vaativa asia sopivan luokan sisään, jolloin luokan purkaja suorittaa tarvittavan siivouksen.

11.5.1 Poikkeukset ja purkajat

C++ pystyy käsittelemään samassa lohkossa vain yhtä poikkeusta kerrallaan. Poikkeuksen heittäminen aiheuttaa tarvittavien staattisen elinkaaren olioiden purkajien suorittamisen *ennen* kuin poikkeus on käsitelty. Jos jo heitetyn poikkeuksen tuloksena kutsutaan purkajaa, joka puolestaan vuotaa ulos oman poikkeuksensa, tulisi samaan aikaan voimaan kaksi poikkeusta. C++:n poikkeuskäsittely ei pysty tähän, joten se kutsuu tällaisesssa tapauksessa suoraan funktiota `terminate` ja lopettaa ohjelman suorituksen. Poikkeuksia purkajien yhteydessä käsitellään tarkemmin aliluvussa 11.8.3.

11.5.2 Poikkeukset ja dynaamisesti luodut oliot

Dynaamisen elinkaaren oliot ovat ongelmallisia. Kuten jo luvussa 3 kerrottiin, **`new`**'llä varattuja olioita ei koskaan tuhota automaattisesti, ja tämä pätee myös poikkeuksen sattuessa. Tilanteen tekee erittäin

vaikeaksi se, että dynaamisesti luotuihin olioihin osoittavat *osoittimet* ovat todennäköisesti normaaleja paikallisia muuttujia, joten ne tuhotaan poikkeuksen seurauksena. Näin muistiin jää helposti tuhoamattomia olioita, joita on mahdoton tuhota, koska niihin ei enää päästä käsiksi.

Ainoa tapa välttää edellä mainitun kaltaisia muistivuotoja on ympäröidä kaikki tarvittavat dynaamisia olioita käsittelevät koodilohkot omalla poikkeuskäsittelijällään. Poikkeuskäsittelijä tuhoaa olion **delete**llä ja sen jälkeen heittää vielä tarvittaessa virheen edelleen ylemmällä tasolla käsiteltäväksi. Listauksessa 11.6 on funktio, joka luo olion dynaamisesti ja tuhoaa sen virhetilanteessa. Huomaa, että koodissa ei riitä varautuminen pelkästään muistin loppumiseen vaan olio täytyy tuhota minkä tahansa muunkin virheen sattuessa.

Jos funktiossa luodaan dynaamisesti useita olioita peräkkäin, on tärkeää, että koodissa varaudutaan siihen, että *muisti loppuu, kun vasta osa olioista on saatu luoduksi*. Jos olioiden luomisen välissä vielä suoritetaan koodia, jossa voi tapahtua virheitä, kannattaa koodiin yleensä kirjoittaa useita sisäkkäisiä valvontalohkoja. Listaus 11.7 seuraavalla sivulla sisältää esimerkin tällaisesta funktiosta.

```
1  void siivousfunktio1()
2  {
3    vector<double>* taulup = new vector<double>();
4
5    try
6    { // Jos täällä sattuu virhe, vektori pitää tuhota
7      keskiarvo2(*taulup);
8    }
9    catch (...)
10   { // Otetaan kiinni kaikki virheet ja tuhotaan vektori
11     delete taulup; taulup = 0;
12     throw; // Heitetään poikkeus edelleen käsiteltäväksi
13   }
14
15   delete taulup; taulup = 0; // Tänne päästään, jos virheitä ei satu
16 }
```

LISTAUS 11.6: Esimerkki dynaamisen olion siivoamisesta

```
 1 void siivousfunktio2()
 2 {
 3    vector<double>* taulup = new vector<double>();
 4    try
 5    {
 6       // Jos täällä sattuu virhe, vektori pitää tuhota
 7       keskiarvo2(*taulup);
 8       vector<double>* taulu2p = new vector<double>();
 9       try
10       { // Jos täällä sattuu virhe, myös uusi vektori pitää tuhota
11          for (unsigned int i = 0; i < taulup->size(); ++i)
12          { // Lasketaan taulukon neliöt
13             taulu2p->push_back((*taulup)[i] * (*taulup)[i]);
14          }
15          cout << "Neliöiden k.a.=" << laskeKeskiarvo(*taulu2p) << endl;
16       }
17       catch (...)
18       { // Otetaan kiinni kaikki virheet ja tuhotaan vektori
19          delete taulu2p; taulu2p = 0;
20          throw; // Heitetään virhe ylemmälle tasolle
21       }
22       delete taulu2p; taulu2p = 0; // Tänne tullaan, jos virheitä ei satu
23    }
24    catch (...)
25    { // Otetaan kiinni kaikki virheet ja tuhotaan vektori
26       delete taulup; taulup = 0;
27       throw; // Heitetään poikkeus edelleen käsiteltäväksi
28    }
29    delete taulup; taulup = 0; // Tänne päästään, jos virheitä ei satu
30 }
```

LISTAUS 11.7: Virheisiin varautuminen ja monta dynaamista oliota

11.6 Poikkeusmääreet

Funktioista ulos vuotavat poikkeukset ovat olennainen osa funktion dokumentaatiota. Ilman niitä funktion kutsuja ei tiedä, mihin kaikkiin poikkeuksiin tulee varautua. C++ antaa mahdollisuuden merkitä funktioon, minkä tyyppiset poikkeukset saavat vuotaa funktiosta ulos. Tämä tapahtuu **poikkeusmääreiden** (*exception specification*) avulla.

Ikävä kyllä käytäntö on standardoinnin jälkeen osoittanut, että hyvästä tarkoituksesta huolimatta poikkeusmääreet eivät C++:ssa ole kovinkaan käyttökelpoinen mekanismi, koska ne suureksi osak-

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