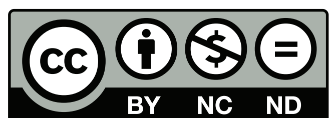
Matti Rintala ja Jyke Jokinen

Olioiden ohjelmointi C++:lla

13. maaliskuuta 2013

© 2012 Matti Rintala ja Jyke Jokinen

(Subversion-revisio 1938)



Tämän teoksen käyttöoikeutta koskee *Creative Commons Nimeä-Ei muutoksia-Epäkaupallinen 1.0 Suomi* -lisenssi.

- **Nimeä** — Teoksen tekijä on ilmoitettava siten kuin tekijä tai teoksen lisensoija on sen määrännyt (mutta ei siten että ilmoitus viittaisi lisenssinantajan tukevan lisenssinsaajaa tai Teoksen käyttötapaa).
- **Ei muutettuja teoksia** — Teosta ei saa muuttaa, muunnella tai käyttää toisen teoksen pohjana.
- **Epäkaupallinen** — Lisenssi ei salli teoksen käyttöä ansiotarkoituksessa.

Lisenssi on nähtävillä kokonaisuudessaan osoitteessa
`http://creativecommons.org/licenses/by-nd-nc/1.0/fi/`

Tämä kirja on taitettu LaTeX-ohjelmistolla (`http://www.tug.org/`). Kaavioiden ja kuvien piirtoon on käytetty XFIG-ohjelmistoa (`http://www.xfig.org/`) ja ohjelmalistaukset on käsitelty LGrind-muotoilijalla.

Sisältö

Esipuhe: Sirkkelin käyttöohje **18**

Alkusanat neljänteen, uudistettuun painokseen **20**

1 Kohti olioita **26**
1.1 Ohjelmistojen tekijöiden ongelmakenttä 26
1.2 Laajojen ohjelmistojen teon vaikeus 27
1.3 Suurten ohjelmistokokonaisuuksien hallinta 28
1.3.1 Ennen kehittyneitä ohjelmointikieliä 29
1.3.2 Tietorakenteet 30
1.3.3 Moduulit 31
1.3.4 Oliot 34
1.3.5 Komponentit 38
1.4 C++: Moduulit käännösyksiköillä 39
1.5 C++: Nimiavaruudet 41
1.5.1 Nimiavaruuksien määritteleminen 42
1.5.2 Näkyvyystarkenninoperaattori 42
1.5.3 Nimiavaruuksien hyödyt 44
1.5.4 **std**-nimiavaruus 45
1.5.5 Standardin uudet otsikkotiedostot 45
1.5.6 Nimiavaruuden synonyymi 46
1.5.7 Lyhyiden nimien käyttö (**using**) 47
1.5.8 Nimeämätön nimiavaruus 49
1.6 Moduulituki muissa ohjelmointikielissä 51
1.6.1 Modula-3 51
1.6.2 Java 51

2 Luokat ja oliot 54
2.1 Olioiden ominaisuuksia 54
2.1.1 Oliolla on tila 55
2.1.2 Olio kuuluu luokkaan 56
2.1.3 Oliolla on identiteetti 57
2.2 Luokan dualismi 58
2.3 C++: Luokat ja oliot 60
2.3.1 Luokan esittely 60
2.3.2 Jäsenmuuttujat 61
2.3.3 Jäsenfunktiot 64
2.3.4 Luokkien ja olioiden käyttö C++:ssa 66

3 Olioiden elinkaari 69
3.1 Olion syntymä 70
3.2 Olion kuolema 71
3.3 Olion elinkaaren määräytyminen 71
3.3.1 Modula-3 72
3.3.2 Smalltalk 74
3.3.3 Java 74
3.3.4 C++ 76
3.4 C++: Rakentajat ja purkajat 79
3.4.1 Rakentajat 80
3.4.2 Purkajat 83
3.5 C++: Dynaaminen luominen ja tuhoaminen 85
3.5.1 **new** 86
3.5.2 **delete** 89
3.5.3 Dynaamisesti luodut taulukot 90
3.5.4 Virheiden välttäminen dynaamisessa luomisessa 90

4 Olioiden rajapinnat 95
4.1 Rajapinnan suunnittelu 96
4.1.1 Hyvän rajapinnan tunnusmerkkejä 97
4.1.2 Erilaisia rajapintoja ohjelmoinnissa 98
4.1.3 Rajapintadokumentaation tuottaminen ja ylläpito 99
4.2 C++: Näkyvyysmääreet 101
4.2.1 **public** 102
4.2.2 **private** 103
4.3 C++: **const** ja vakio-oliot 105
4.3.1 Perustyyppiset vakiot 105

4.3.2 Vakio-oliot 106
4.3.3 Vakioviitteet ja -osoittimet 109
4.4 C++: Luokan ennakkoesittely 112
4.4.1 Ennakkoesittely kapseloinnissa 113

5 Oliosuunnittelu 116
5.1 Oliosuunnittelua ohjaavat ominaisuudet 117
5.1.1 Mitä suunnittelu on? 117
5.1.2 Abstraktio ja tiedon kätkentä 118
5.1.3 Osien väliset yhteydet ja lokaalisuusperiaate . . 118
5.1.4 Laatu 120
5.2 Oliosuunnittelun aloittaminen 121
5.2.1 Luokan vastuualue 122
5.2.2 Kuinka löytää luokkia? 123
5.2.3 CRC-kortti 124
5.2.4 Luokka, attribuutti vai operaatio? 126
5.3 Oliosuunnitelman graafinen kuvaus 126
5.3.1 UML:n historiaa 127
5.3.2 Luokat, oliot ja rajapinnat 128
5.3.3 Luokkien väliset yhteydet 130
5.3.4 Ajoaikaisen käyttäytymisen kuvaamistapoja . . 138
5.3.5 Ohjelmiston rakennekuvaukset 140
5.4 Saatteeksi suunnitteluun 140

6 Periytyminen 142
6.1 Periytyminen, luokkahierarkiat, polymorfismi 143
6.2 Periytyminen ja uudelleenkäyttö 147
6.3 C++: Periytymisen perusteet 149
6.3.1 Periytyminen ja näkyvyys 150
6.3.2 Periytyminen ja rakentajat 152
6.3.3 Periytyminen ja purkajat 154
6.3.4 Aliluokan olion ja kantaluokan suhde 155
6.4 C++: Periytymisen käyttö laajentamiseen 156
6.5 C++: Virtuaalifunktiot ja dynaaminen sitominen 159
6.5.1 Virtuaalifunktiot 159
6.5.2 Dynaaminen sitominen 160
6.5.3 Olion tyypin ajoaikainen tarkastaminen 164
6.5.4 Ei-virtuaalifunktiot ja peittäminen 167
6.5.5 Virtuaalipurkajat 168

6.5.6 Virtuaalifunktioiden hinta 169
6.5.7 Virtuaalifunktiot rakentajissa ja purkajissa . . . 170
6.6 Abstraktit kantaluokat 172
6.7 Moniperiytyminen 175
6.7.1 Moniperiytymisen idea 175
6.7.2 Moniperiytyminen eri oliokielissä 176
6.7.3 Moniperiytymisen käyttökohteita 177
6.7.4 Moniperiytymisen vaaroja 178
6.7.5 Vaihtoehtoja moniperiytymiselle 179
6.8 C++: Moniperiytyminen 180
6.8.1 Moniperiytyminen ja moniselitteisyys 181
6.8.2 Toistuva moniperiytyminen 186
6.9 Periytyminen ja rajapintaluokat 190
6.9.1 Rajapintaluokkien käyttö 191
6.9.2 C++: Rajapintaluokat ja moniperiytyminen 193
6.9.3 Ongelmia rajapintaluokkien käytössä 195
6.10 Periytyminen vai kooste? 198
6.11 Sovelluskehykset 199

7 Lisää olioiden elinkaaresta 201
7.1 Olioiden kopiointi 202
7.1.1 Erilaiset kopiointitavat 203
7.1.2 C++: Kopiorakentaja 206
7.1.3 Kopiointi ja viipaloituminen 210
7.2 Olioiden sijoittaminen 215
7.2.1 Sijoituksen ja kopioinnin erot 215
7.2.2 C++: Sijoitusoperaattori 216
7.2.3 Sijoitus ja viipaloituminen 221
7.3 Oliot arvoparametreina ja paluuarvoina 224
7.4 Tyyppimuunnokset 227
7.4.1 C++:n tyyppimuunnosoperaattorit 228
7.4.2 Ohjelmoijan määrittelemät tyyppimuunnokset 232
7.5 Rakentajat ja **`struct`** 237

8 Lisää rajapinnoista 240
8.1 Sopimus rajapinnasta 240
8.1.1 Palveluiden esi- ja jälkiehdot 241
8.1.2 Luokkainvariantti 243
8.1.3 Sopimussuunnittelun käyttö 244

8.1.4 C++: luokan sopimusten tarkastus 244
8.2 Luokan rajapinta ja olion rajapinta 248
8.2.1 Metaluokat ja luokkaoliot 249
8.2.2 C++: Luokkamuuttujat 250
8.2.3 C++: Luokkafunktiot 252
8.3 Tyypit osana rajapintaa 253
8.4 Ohjelmakomponentin sisäiset rajapinnat 258
8.4.1 C++: Ystäväfunktiot 259
8.4.2 C++: Ystäväluokat 260

9 Geneerisyys 263
9.1 Yleiskäyttöisyys, pysyvyys ja vaihtelevuus 264
9.2 Suunnittelun geneerisyys: suunnittelumallit 267
9.2.1 Suunnittelumallien edut ja haitat 267
9.2.2 Suunnittelumallin rakenne 268
9.3 Valikoituja esimerkkejä suunnittelumalleista 271
9.3.1 Kokoelma (*Composite*) 271
9.3.2 Iteraattori (*Iterator*) 273
9.3.3 Silta (*Bridge*) 274
9.3.4 C++: Esimerkki suunnittelumallin toteutuksesta 276
9.4 Geneerisyys ja periytymisen rajoitukset 277
9.5 C++: Toteutuksen geneerisyys: mallit (*template*) 283
9.5.1 Mallit ja tyyppiparametrit 284
9.5.2 Funktiomallit 286
9.5.3 Luokkamallit 287
9.5.4 Tyyppiparametreille asetetut vaatimukset . . . 290
9.5.5 Erilaiset mallien parametrit 292
9.5.6 Mallien erikoistus 295
9.5.7 Mallien ongelmia ja ratkaisuja 297

10 Geneerinen ohjelmointi: STL ja metaohjelmointi 302
10.1 STL:n perusperiaatteita 303
10.1.1 STL:n rakenne 304
10.1.2 Algoritmien geneerisyys 305
10.1.3 Tietorakenteiden jaotteluperusteet 306
10.1.4 Tehokkuuskategoriat 308
10.2 STL:n säiliöt 310
10.2.1 Sarjat (“peräkkäissäiliöt”) 312
10.2.2 Assosiatiiviset säiliöt 317

10.2.3 Muita säiliöitä 323
10.3 Iteraattorit 326
10.3.1 Iteraattoreiden käyttökohteet 327
10.3.2 Iteraattorikategoriat 329
10.3.3 Iteraattorit ja säiliöt 332
10.3.4 Iteraattoreiden kelvollisuus 334
10.3.5 Iteraattorisovittimet 336
10.4 STL:n algoritmit 337
10.5 Funktio-oliot 340
10.5.1 Toiminnallisuuden välittäminen algoritmille . . 341
10.5.2 Funktio-olioiden periaate 343
10.5.3 STL:n valmiit funktio-oliot 347
10.6 C++: Template-metaohjelmointi 348
10.6.1 Metaohjelmoinnin käsite 349
10.6.2 Metaohjelmointi ja geneerisyys 351
10.6.3 Metafunktiot 352
10.6.4 Esimerkki metafunktioista: numeric_limits . . 357
10.6.5 Esimerkki metaohjelmoinnista: tyypin valinta . 360
10.6.6 Esimerkki metaohjelmoinnista: optimointi . . . 362

11 Virhetilanteet ja poikkeukset 366
11.1 Mikä virhe on? 367
11.2 Mitä tehdä virhetilanteessa? 369
11.3 Virhehierarkiat 371
11.4 Poikkeusten heittäminen ja sieppaaminen 374
11.4.1 Poikkeushierarkian hyväksikäyttö 374
11.4.2 Poikkeukset, joita ei oteta kiinni 377
11.4.3 Sisäkkäiset valvontalohkot 378
11.5 Poikkeukset ja olioiden tuhoaminen 380
11.5.1 Poikkeukset ja purkajat 380
11.5.2 Poikkeukset ja dynaamisesti luodut oliot 380
11.6 Poikkeusmääreet 382
11.7 Muistivuotojen välttäminen: auto_ptr 383
11.7.1 Automaattiosoittimet ja muistinhallinta 384
11.7.2 Automaattiosoittimien sijoitus ja kopiointi . . . 385
11.7.3 Automaattiosoittimien käytön rajoitukset 387
11.8 Olio-ohjelmointi ja poikkeusturvallisuus 388
11.8.1 Poikkeustakuut 389
11.8.2 Poikkeukset ja rakentajat 393

11.8.3 Poikkeukset ja purkajat 397
11.9 Esimerkki: poikkeusturvallinen sijoitus 399
11.9.1 Ensimmäinen versio 399
11.9.2 Tavoitteena vahva takuu 401
11.9.3 Lisätään epäsuoruutta 403
11.9.4 Tilan eriyttäminen ("*pimpl*"-idiomi) 405
11.9.5 Tilan vaihtaminen päikseen 407

Liite A. C++: Ei-olio-ominaisuuksia 410
A.1 Viitteet 410
A.2 **inline** 413
A.3 vector 414
A.4 string 416

Liite B. C++-tyyliopas 420

Kirjallisuutta 437

Englanninkieliset termit 444

Listaukset

1.1 Päiväysmoduulin tietorakenne ja osa rajapintaa . . . 34
1.2 Päiväysmoduulin käyttöesimerkki 35
1.3 Päiväysolioiden käyttö 36
1.4 Päiväysmoduulin esittely C-kielellä, paivays.h 39
1.5 Nimiavaruudella toteutettu rajapinta 43
1.6 Rajapintafunktioiden toteutus erillisessä tiedostossa . 43
1.7 Nimiavaruuden rakenteiden käyttäminen 44
1.8 C-kirjastofunktiot C++:n nimiavaruudessa 46
1.9 Nimiavaruuden synonyymi (aliasointi) 47
1.10 **using**-lauseen käyttö 48
1.11 **using**-lauseen käyttö eri otsikkotiedostojen kanssa . . 49
1.12 Nimeämätön nimiavaruus 50
1.13 Moduulituki Modula-3-kielessä 52
1.14 Moduulituki Java-kielessä 52

2.1 Esimerkki luokan esittelystä, pienipaivays.hh 62
2.2 Jäsenfunktioiden toteutus, pienipaivays.cc 65
2.3 Esimerkki luokan käytöstä, ppkaytto.cc 68

3.1 Esimerkki olion elinkaaresta Modula-3:lla 73
3.2 Esimerkki olion elinkaaresta Smalltalkilla 75
3.3 Esimerkki olion elinkaaresta Javalla 76
3.4 Esimerkki olion elinkaaresta C++:lla 78
3.5 Paivays-luokan rakentaja 80
3.6 Esimerkki rakentajasta olion ollessa jäsenmuuttujana 82
3.7 Paivays-luokan purkaja 84
3.8 Esimerkki olion dynaamisesta luomisesta **new**'llä . . . 88
3.9 Taulukko, joka omistaa sisältämänsä kokonaisluvut . 92

3.10 Taulukko, joka ei omista sisältämiään kokonaislukuja 93

4.1 Jäsenfunktio, joka palauttaa viitteen jäsenmuuttujaan 104
4.2 Pääsy toisen saman luokan olion **private**-osaan . . . 105
4.3 Päiväysluokka **const**-sanoineen 107
4.4 Esimerkki virheestä, kun **const**-sana unohtuu 111
4.5 Esimerkki ennakkoesittelystä 113
4.6 Ennakkoesittely kapseloinnissa 114

5.1 Lainausjärjestelmän esittely, `lainausjarjestelma.hh` . 134

6.1 Periytymisen syntaksi C++:lla 150
6.2 Periytyminen ja rakentajat 154
6.3 Kirjan tiedot muistava luokka 157
6.4 Kirjaston kirjan palvelut tarjoava aliluokka 158
6.5 Luokan `Kirja` virtuaalifunktiot 161
6.6 Luokan `KirjastonKirja` virtuaalifunktiot 162
6.7 Dynaaminen sitominen C++:ssa 163
6.8 Olion tyypin ajoaikainen tarkastaminen 165
6.9 Esimerkki **typeid**-operaattorin käytöstä 167
6.10 Abstrakteja kantaluokkia ja puhtaita virtuaalifunktioita 173
6.11 Puhdas virtuaalifunktio, jolla on myös toteutus 174
6.12 Moniselitteisyyden yksi välttämistapa 182
6.13 Jäsenfunktiokutsun moniselitteisyyden eliminointi . 184
6.14 Moniselitteisyyden eliminointi väliluokilla 185
6.15 Erilliset rajapinnat Javassa 192
6.16 Rajapintaluokkien toteutus moniperiytymisellä C++:ssa 193
6.17 Rajapintaluokan purkaja esittelyyn upotettuna 195

7.1 Esimerkki kopiorakentajasta 207
7.2 Kopiorakentaja aliluokassa 208
7.3 Viipaloitumisen kiertäminen `kloonaa`-jäsenfunktiolla 214
7.4 Esimerkki sijoitusoperaattorista 217
7.5 Sijoitusoperaattori periytetyssä luokassa 220
7.6 Viipaloitumismahdollisuudet sijoituksessa 222
7.7 Viipaloitumisen estäminen ajoaikaisella tarkastuksella 223
7.8 Olio arvoparametrina ja paluuarvona 225
7.9 Esimerkki **const_cast**-muunnoksesta 230
7.10 Tiedon esitystavan muuttaminen ja **reinterpret_cast** 233

7.11 Esimerkki rakentajasta tyyppimuunnoksena 234
7.12 Esimerkki muunnosjäsenfunktiosta 237
7.13 **struct**-tietorakenne, jossa on olioita 237
7.14 **struct**, jolla on rakentaja 239

8.1 Varmistusrutiinin toteutus 247
8.2 Esimerkki luokkainvariantin toteutuksesta jäsenfunktiona 248
8.3 Esimerkki luokkamuuttujien ja -funktioiden käytöstä 254
8.4 Tyypillinen päiväysluokan esittely 255
8.5 Parannettu päiväysluokan esittely 257
8.6 Luokan määrittelemä tyyppi paluutyyppinä 258
8.7 Laajemman rajapinnan salliminen ystäväfunktioille . 261
8.8 Ystäväluokat 262

9.1 Virhetiedoterajapinnan esittely ja toteutus 278
9.2 Toteutusten kantaluokka, `virheikkunatoteutus.hh` . . 279
9.3 Toteutus virheikkunoista, `virheikkunaversiot.hh` . . 279
9.4 Eri virheikkunatoteutusten valinta 279
9.5 Parametreista pienemmän palauttava funktiomalli . . 286
9.6 Tietotyypin "pari" määrittelevä luokkamalli 288
9.7 Esimerkki luokkamallin `Pari` jäsenfunktioista 289
9.8 Luokkamallin sisällä oleva jäsenfunktiomalli 290
9.9 Parempi versio listauksen 9.5 funktiomallista 292
9.10 Mallin oletusparametrit 293
9.11 Malli, jolla on vakioparametri 294
9.12 Mallin malliparametri 295
9.13 Luokkamallin `Pari` erikoistus totuusarvoille 296
9.14 Funktiomallin `min` erikoistus päiväyksille 297
9.15 Luokkamallin `Pari` osittaiserikoistus 297
9.16 Avainsanan **export** käyttö 299
9.17 Tyypin määrääminen avainsanalla **typename** 300

10.1 Fibonaccin luvut `vector`illa 314
10.2 Puskurin toteutus `deque`lla 316
10.3 Nimien rekisteröinti `set`illä 319
10.4 Tehokkaampi versio listauksesta 10.3 320
10.5 Nimirekisteröinti `multiset`illä 320
10.6 Nimirekisteröinti `map`illä 321

10.7 multimapillä toteutettu puhelinluetteloluokka 323
10.8 Puhelinluettelon toteutus 324
10.9 Säiliön läpikäyminen iteraattoreilla 333
10.10 Esimerkki STL:n algoritmien käytöstä 340
10.11 Funktio-osoittimen välittäminen parametrina 342
10.12 Esimerkki funktio-olioluokasta 344
10.13 Funktio-olion käyttöesimerkkejä 345
10.14 Funktio-olion käyttö STL:ssä 346
10.15 C++:n funktio-olioiden less ja bind2nd käyttö 349
10.16 Esimerkki yksinkertaisesta metaohjelmoinnista 353
10.17 Yksinkertainen trait-metafunktio 354
10.18 Erikoistamalla tehty template-metafunktio 355
10.19 Osittaiserikoistuksella tehty metafunktio 356
10.20 Esimerkki numeric_limits-metafunktion käytöstä . . 359
10.21 Metafunktion IF toteutus 361
10.22 Metafunktion IF käyttöesimerkki 361
10.23 Esimerkki metaohjelmoinnista optimoinnissa 364

11.1 Virhetyypit C++:n luokkina 373
11.2 Esimerkki omasta virheluokasta 374
11.3 Esimerkki C++:n poikkeuskäsittelijästä 376
11.4 Virhekategorioiden käyttö poikkeuksissa 377
11.5 Sisäkkäiset valvontalohkot 379
11.6 Esimerkki dynaamisen olion siivoamisesta 381
11.7 Virheisiin varautuminen ja monta dynaamista oliota . 382
11.8 Esimerkki automaattiosoittimen auto_ptr käytöstä . . 385
11.9 Automaattiosoitin ja omistuksen siirto 386
11.10 Esimerkki luokasta, jossa on useita osaolioita 394
11.11 Olioiden dynaaminen luominen rakentajassa 395
11.12 Funktion valvontalohko rakentajassa 397
11.13 Yksinkertainen luokka, jolla on sijoitusoperaattori . . 399
11.14 Sijoitus, joka pyrkii tarjoamaan vahvan takuun (ei toimi) 402
11.15 Kirjaluokka epäsuoruuksilla 403
11.16 Uuden kirjaluokan rakentaja, purkaja ja sijoitus . . . 404
11.17 Kirja eriytetyllä tilalla ja automaattiosoittimella . . . 406
11.18 Eriytetyn tilan rakentajat, purkaja ja sijoitus 406
11.19 Kirja, jossa on nothrow-vaihto 408
11.20 Yhdistelmä tilan eriyttämisestä ja vaihdosta 409

A.1 Osamerkkijonon “ja” etsintä (C++) 417
A.2 Osamerkkijonon “ja” etsintä (C) 418

Kuvat

1.1 Tyypillinen assembly-ohjelman spagettirakenne . . . 30
1.2 Päiväykset tietorakenteina 31
1.3 Päiväyksiä käsittelevän moduulin rakenne 32
1.4 Rajapinta osana tietorakennetta (oliot) 35
1.5 Sama otsikkotiedosto voi tulla käyttöön useita kertoja 40
1.6 Moduulirakenne C-kielellä 41

2.1 Ongelmasta ohjelmaksi olioilla 55
2.2 Päiväys-luokka ja siitä tehdyt oliot A ja B 59

4.1 Väärin tehty luokkien keskinäinen esittely (ei toimi) . 112
4.2 Oikein tehty luokkien keskinäinen esittely 115

5.1 Erilaisia yhteysvaihtoehtoja kuuden moduulin välillä 119
5.2 Moduulien välisiä yksisuuntaisia riippuvuuksia . . . 120
5.3 Esimerkki käyttötapauksesta 124
5.4 Kuva keskeneräisen CRC-korttipelin yhdestä kortista . 125
5.5 Luokka, olio ja rajapinta 129
5.6 Tarkennetun suunnitelman luokka ja näkyvyysmääreitä 130
5.7 UML:n yhteystyyppejä 131
5.8 Luokka “Heippa” käyttää Javan grafiikkaolioita 132
5.9 UML-assosiaatioiden lukumäärämerkintöjä 132
5.10 Esimerkki luokkien välisistä assosiaatioista 133
5.11 UML:n koostesuhteita 135
5.12 Osaston, sen laitosten ja työntekijöiden välisiä yhteyksiä 136
5.13 Kirjasta periytetty luokka, joka toteuttaa kaksi rajapintaa 137

5.14 Palautuspäivämäärän asettamisen tapahtumasekvenssi 138
5.15 Kirjastokortin tiloja 139

6.1 Periytymiseen liittyvää terminologiaa 144
6.2 Eliöitä mallintavan ohjelman periytymishierarkiaa . . 145
6.3 Näkyvyyttä kuvaava kantaluokka ja periytettyjä luokkia 148
6.4 Periytymishierarkia ja oliot [Koskimies, 2000] 149
6.5 Periytyminen ja näkyvyys 151
6.6 Moniperiytyminen ja sen vaikutus 176
6.7 Toistuva moniperiytyminen 187
6.8 Toistuva moniperiytyminen ja olion rakenne 188
6.9 Rakentajat virtuaalisessa moniperiytymisessä 189
6.10 Luokat, jotka toteuttavat erilaisia rajapintoja 191
6.11 Ongelma yhdistettyjen rajapintojen kanssa 197
6.12 Insinööri, viulisti ja isä 199
6.13 Aliohjelmakirjasto ja sovelluskehys 200

7.1 Viitekopiointi 203
7.2 Matalakopiointi 204
7.3 Syväkopiointi 205
7.4 Viipaloituminen olion kopioinnissa 211
7.5 Viipaloituminen olioiden sijoituksessa 223
7.6 Oliot arvoparametreina ja paluuarvoina 226

8.1 Luokka- ja metaluokkahierarkiaa Smalltalkissa 250

9.1 Pysyvyys- ja vaihtelevuusanalyysi ja yleiskäyttöisyys 266
9.2 Kokoelman toteuttava luokka 269
9.3 Taulukko joka sisältää kokoelmia 270
9.4 Suunnittelumallin UML-symboli ja sen käyttö 271
9.5 Suunnittelumalli Kokoelma 272
9.6 Suunnittelumalli Iteraattori 274
9.7 Suunnittelumalli Silta 276
9.8 Yleiskäyttöisen taulukon toteutus periyttämällä . . . 281

10.1 Tietorakenteiden herättämiä mielikuvia 306
10.2 Erilaisia tehokkuuskategorioita 310
10.3 Sarjojen operaatioiden tehokkuuksia 313
10.4 Iteraattorit ja säiliöt 328
10.5 Iteraattorikategoriat 330

10.6 Funktio-olioiden idea 345
10.7 Funktio-olio bind2nd(less<int>(), 3) 348
10.8 Metafunktion numeric_limits palvelut 358

11.1 C++-standardin virhekategoriat 372

A.1 Esimerkkien viittaukset 412
A.2 C-taulukon ja vektorin vertailu 416
A.3 Merkkijonotaulukon ja C++:n stringin vertailu 417

Esipuhe: Sirkkelin käyttöohje

Siirtyminen C-kielestä C++-kieleen käynnistyi pikkuhiljaa jo 1980-luvun loppupuolella. Olin tavattoman innostunut tästä, koska näin sen mahdollisuutena siirtää tutkimuslaboratorioissa syntyneitä oliokeskeisiin menetelmiin liittyviä asioita osaksi arkipäiväistä ohjelmistotyötä. 1990-luvun alkupuoli olikin ohjelmistoteollisuudessa oliomenetelmien käyttöönoton aikaa, ja hyvin usein käyttöönotto tapahtui juuri C++-kieleen siirtymisen kautta.

Yleinen ensivaikutelma C++-kielestä oli, että se on oliopiirteillä laajennettu C-kieli ja vain hieman C-kieltä monimutkaisempi. Jo ensimmäiset käytännön kokemukset kielestä kuitenkin osoittivat, että tämä oli näköharhaa. Käytännössä C++-kieleen lisättyjen olio-ominaisuuksien aiheuttamat implikaatiot ovat paljon laajempia kuin aluksi osattiin ennakoida. Tätä todistavat esimerkiksi yritysten ohjelmistokehitystä varten laatimat tyylioppaat: C-tyyliopas saattaa olla vain muutamien sivujen mittainen kokoelma sääntöjä, kun pisimmät C++-oppaat lähentelevät sataa sivua.

Käytäntö on osoittanut, että C++-kieli sisältää ominaisuudet, joilla olio-ohjelmoinnin lupaukset muuan muassa uudelleenkäytettävyyden ja ylläpidettävyyden osalta pystytään suurelta osin lunastamaan. Toisaalta olio-ominaisuuksien taitamattomasta käytöstä saattaa aiheutua isoja ongelmia. C++ on kuin tehokas sirkkeli ilman suojavälineitä: ammattilaisen käsissä se on tuottava ja varmatoiminen työkalu, mutta noviisin käsissä vaarallinen ase. Oman kokemukseni mukaan tehosirkkeli-noviisi-yhdistelmä ei ole tämän päivän ohjelmistotuotannossa harvinainen yhdistelmä, ja pätevälläkin ammattilaisella

on vielä paljon opittavaa.

C++-kieltä käsitteleviä oppikirjoja on kasapäin. Useimmat kirjat lähestyvät aihetta esittelemällä kielen ominaisuuksia, kertomatta miten ja miksi niitä käytetään. Tässä kirjassa keskitytään kielen syntaksin sijasta juuri näihin C++-sirkkelin käyttöön liittyviin miten ja miksi -kysymyksiin. Se on antoisaa luettavaa niin noviisille kuin vähän vanhemmallekin konkarille. Harvoinpa kirjalle on olemassa näin selkeä tilaus niin oppilaitoksissa kuin teollisuudessakin.

Tampereella 3.9.2000, prof. *Ilkka Haikala*

Alkusanat neljänteen, uudistettuun painokseen

Oliokeskeisyyteen liittyvässä markkinahumussa termistä oliokeskeinen *on muodostunut vähitellen termin* hyvä *synonyymi; lähes mikä hyvänsä idea tai tuote voidaan naamioida oliokeskeiseksi myyntitarkoituksissa. Oleellisen erottaminen epäoleellisesta tulee tämän myötä yhä hankalammaksi.*

– Ilkka Haikala [Haikala ja Märijärvi, 2002]

Object-oriented programming is an exceptionally bad idea which could only have originated in California.

– Edsger Wybe Dijkstra

Olio-ohjelmointi on nopeasti muuttunut “uudesta ja ihmeellisestä” ohjelmointitavasta arkipäiväiseksi valtavirran ohjelmointimenetelmäksi. Samaan aikaan aiheeseen liittyviä kirjoja on julkaistu lukemattomia määriä.

Useissa olio-ohjelmointia käsittelevissä kirjoissa on kuitenkin tyypillisesti yksi ongelma. Osa kirjoista on kirjoitettu hyvin yleisellä tasolla, ja ne keskittyvät oliosuunnitteluun ja olio-ohjelmoinnin teoriaan paneutumatta siihen, miten nämä asiat toteutetaan käytännön olio-ohjelmointia tukevissa kielissä. Toiset kirjat ovat puolestaan lähes yksinomaan jonkin oliokielen oppikirjoja, jolloin ne usein keskittyvät lähinnä tietyn kielen syntaksin ja erikoisuuksien opettamiseen.

Tämä kirja pyrkii osumaan näiden kahden ääripään puoliväliin. Siinä pyritään antamaan mahdollisimman monesta olio-ohjelmoinnin aiheesta sekä yleinen ("teoreettinen" olisi ehkä liian mahtipontinen sana) että käytännönläheinen kuva. Kirjan käytännön puolessa keskitytään etupäässä C++-kieleen, koska se on tällä hetkellä yleisin olio-ohjelmointia tukevista kielistä. Myös joidenkin muiden oliokielten ominaisuuksia käydään kuitenkin läpi silloin, kun ne oleellisesti eroavat C++:n oliomallista.

Kirjan päämäärän johdosta tämä teos ***ei ole ohjelmoinnin alkeisopas***. Se edellyttää lukijaltaan "perinteisen" ei-olio-ohjelmoinnin perustaitoja sekä C- tai C++-kielen alkeiden tuntemista. Olio-ohjelmoinnista lukijan ei kuitenkaan tarvitse tietää mitään ennen tämä kirjan lukemista.

Tämä teos ei myöskään pyri olemaan itse C++-kielen oppikirja. Se esittelee suurimman osan C++:n olio-ohjelmointiin liittyvistä piirteistä ja käsittelee niiden käyttöä olio-ohjelmoinnissa, mutta näiden piirteiden kaikkiin yksityiskohtiin ei ole mahdollista paneutua kirjan puitteissa. Tämän vuoksi C++:aa opettelevan kannattaa hankkia tämän kirjan rinnalle käsikirjaksi jokin C++-kielen oppikirja, josta voi etsiä vastauksia itse kielen omituisuuksiin liittyviin kysymyksiin. Kirjaksi kelpaa esim. "The C++ Programming Language" [Stroustrup, 1997] (myös suomennettuna [Stroustrup, 2000]) tai jokin muu monista muista vaihtoehdoista.

Kirjan materiaali on saanut alkunsa tekijöiden Tampereen teknillisellä korkeakoululla luennoimasta kurssista Olio-ohjelmointi sekä lukuisista yrityksille pidetyistä C++-kursseista. Kirjaan liittyviä uutisia ja lisätietoja löytyy WWW-sivulta `http://www.cs.tut.fi/~oliot/kirja/`. Sivun kautta voi lähettää myös palautetta kirjan tekijöille.

Olemme käyttäneet kirjaa myös omassa opetuksessamme TTY:llä ja yrityksissä. Näiden kurssien myötä olemme tehneet kirjaa varten opetuskalvosarjan, jonka toimitamme mielellämme halukkaille (yhteystietomme löytyvät kirjan kotisivulta).

Otamme mielellämme vastaan kaikki kirjaa koskevat kommentit ja kehitysehdotukset. Kumpikin kirjoittaja mielellään syytää toista kaikista kirjaan vielä mahdollisesti jääneistä virheistä.

Neljäs, uudistettu painos

Opetus on kehittyvä prosessi. Tampereen teknillisellä yliopistolla on ohjelmistotekniikan opetuksessa nykyisin kaksi kurssia olio-ohjelmoinnista (perus- ja jatkokurssi). Erityisesti jatkokurssin tarpeiden mukaisesti olemme laajentaneet edellisiä painoksia mm. seuraavissa aiheissa: C++:n nimiavaruudet, moniperiytyminen, viipaloituminen kopioinnissa ja sijoituksessa, geneerisyys ja template-metaohjelmointi sekä poikkeusturvallisuus. Edelleen painotus on yleisissä olio-ohjelmoinnin periaatteissa ja niiden soveltamisessa C++-ohjelmointikielellä.

Edellä mainitut laajemmat lisäykset tehtiin kirjan vuoden 2003 kolmanteen, uudistettuun painokseen. Nyt vuonna 2005 julkaistiin kirjan neljäs painos. Siinä on kolmanteen painokseen verrattuna tehty joitakin korjauksia, lisäyksiä, päivityksiä ja tyylillisiä parannuksia.

Kirjan rakenne

Tämä kirja on *tarkoitettu luettavaksi järjestyksessä alusta loppuun,* ja se esittelee olio-ohjelmoinnin ominaisuuksia sellaisessa järjestyksessä, joka on ainakin tekijöiden kursseilla tuntunut toimivalta. Kirjan lukujen rakenne on seuraava:

1. **Kohti olioita.** Luku toimii johdantona olioajatteluun. Siinä käydään läpi ongelmia, jotka ilmenevät suurten ohjelmistojen tekemisessä, sekä puhutaan *modulaarisuudesta* olio-ohjelmoinnin “esi-isänä”. Modulaarisuuden yhteydessä esitellään myös C++:n *nimiavaruudet (namespace).*

2. **Luokat ja oliot.** Tämä luku käy läpi olio-ohjelmoinnin tärkeimmät käsitteet, *luokat* ja *oliot*. Lisäksi luvussa opetetaan C++:n oliomallin perusteet.

3. **Olioiden elinkaari.** Tässä luvussa kerrotaan olioiden luomiseen ja tuhoamiseen liittyvistä ongelmista ja esitellään, miten nämä ongelmat on ratkaistu eri oliokielissä.

4. **Olioiden rajapinnat.** Luvussa käsitellään rajapintojen suunnittelun tärkeimpiä periaatteita ja kerrotaan, millaisia erilaisia rajapintoja olio voi C++:ssa tarjota käyttäjilleen.

5. **Oliosuunnittelu.** Luku käy läpi tärkeimpiä oliosuunnittelun menetelmiä, muiden muassa *UML:ää.*

6. **Periytyminen.** *Periytyminen* on ehkä tärkeimpiä olio-ohjelmoinnin tuomia uusia asioita. Tämä luku käsittelee periytymisen teoriaa yleisesti sekä sen toteutusta C++-kielessä. Tässä uudistetussa painoksessa luku käsittelee myös *moniperiytymistä* ja sen suhdetta *rajapintaluokkiin*.

7. **Lisää olioiden elinkaaresta.** Tässä luvussa käydään läpi lisää olioiden elinkaareen liittyviä asioita, mm. *kopioiminen, sijoittaminen, viipaloituminen (slicing)* ja *tyyppimuunnokset.*

8. **Lisää rajapinnoista.** Vastaavasti tämä luku sisältää lisätietoa rajapintoihin liittyvistä aiheista, kuten *sopimussuunnittelusta,* luokkatason rajapinnasta, rajapintatyypeistä sekä komponentin sisäisistä rajapinnoista.

9. **Geneerisyys.** Luvussa käsitellään yleiskäyttöisyyden ja uudelleenkäytettävyyden periaatteita. Suunnittelun geneerisyydestä esitellään *suunnittelumallit* ja C++:n toteutuksen geneerisyydestä *mallit (template)*. Lisäksi luvussa pohditaan periytymisen ja geneerisyyden suhdetta.

10. **Geneerinen ohjelmointi — STL ja metaohjelmointi.** Tämä luku esittelee C++:n STL-kirjaston esimerkkinä geneerisestä ohjelmakirjastosta. Lisäksi luku antaa katsauksen *template-metaohjelmointiin* esimerkkinä mukautuvista geneerisistä mekanismeista.

11. **Virhetilanteet ja poikkeukset.** Viimeisessä luvussa käydään vielä läpi *poikkeukset* (C++:n tapa virhetilanteiden käsittelyyn) siinä laajuudessa, kuin ne liittyvät olio-ohjelmointiin. Luku käsittelee myös olioiden *poikkeusturvallisuutta* ja sen toteuttamista C++:lla.

Lisäksi liitteessä A esitellään lyhyesti sellaisia C++:n ei-olio-ominaisuuksia, joita ei ole ollut C-kielessä, mutta joita käytetään tämän kirjan esimerkeissä. Liitteessä A käydään läpi

- viitetyypit

- `inline`-sanan käyttö tehokkuusoptimoinnissa
- `vector`-taulukot C:n taulukkojen korvaajina
- `string`-merkkijonot C:n `char*`-merkkijonojen korvaajina.

Liitteen tarkoituksena on että lukija, joka ei näitä ominaisuuksia tunne, saa niistä liitteen avulla sellaisen käsityksen, että kirjan koodiesimerkkien lukeminen onnistuu vaivatta. Kyseisiä ominaisuuksia ei kuitenkaan opeteta liitteessä perinpohjaisesti.

Liite B sisältää C++-tyylioppaan. Siihen on kerätty tekijöiden mielestä tärkeitä ohjelmointityyliin ja C++:n sudenkuoppien välttämiseen liittyviä ohjeita selityksineen. Tämä tyyliopas on käytössä mm. Tampereen teknillisen yliopiston Ohjelmistotekniikan laitoksella opetuksessa ja ohjelmointiprojekteissa.

Kirjan lopussa on myös kirjallisuusluettelo sekä aakkosellinen luettelo englanninkielisistä olio-ohjelmoinnin termeistä ja niiden suomenkielisistä vastineista tässä teoksessa (itse tekstissä jokaisesta uudesta termistä on pyritty kertomaan sekä suomenkielinen että englanninkielinen sana). Lisäksi kirjan loppuun on liitetty normaali aakkosellinen hakemisto.

Kiitokset

Haluamme kiittää Tampereen teknillisen yliopiston Ohjelmistotekniikan laitoksen henkilökuntaa ja erityisesti sen professoreita *Reino Kurki-Suonio* ja *Ilkka Haikala*, jotka ovat luoneet opetuksellaan, esimerkillään ja olemuksellaan työympäristön, jossa on todella ilo työskennellä.

Kirjan käsikirjoituksen kommentoinnista suurkiitokset professoreille *Ilkka Haikala*, *Hannu-Matti Järvinen*, *Kai Koskimies*, *Reino Kurki-Suonio* ja *Markku Sakkinen* sekä kollegoillemme *Kirsti Ala-Mutka*, *Joni Helin*, *Vespe Savikko* ja *Antti Virtanen*.

Kiitokset kuuluvat myös olio-ohjelmointi- ja C++-kurssiemme opiskelijoille, jotka ovat kommenteillaan ja kysymyksillään vaikuttaneet opetuksemme ja materiaalimme sisältöön.

Lopuksi haluamme vielä kiittää kaikkia perheenjäseniämme, sukulaisiamme, ystäviämme ja tuttaviamme, joiden ansiosta (tai joista huolimatta) olemme säilyneet edes jotenkin tervejärkisinä kirjoitusurakan aikana.

Tampereella 15.4.2005

Matti Rintala

Jyke Jokinen

Luku 1

Kohti olioita

Trurl päätti lopulta vaientaa hänet kerta kaikkiaan rakentamalla koneen, joka osaisi kirjoittaa runoja. Ensin Trurl keräsi kahdeksansataakaksikymmentä tonnia kybernetiikkaa käsittelevää kirjallisuutta ja kaksitoistatuhatta tonnia parasta runoutta, istahti aloilleen ja luki kaiken läpi. Aina kun hänestä alkoi tuntua, ettei hän pystyisi nielaisemaan enää ainuttakaan kaavakuvaa tai yhtälöä, hän vaihtoi runouteen, ja päinvastoin. Jonkin ajan kuluttua hänelle alkoi valjeta, että koneen rakentaminen olisi lastenleikkiä verrattuna sen ohjelmoimiseen. Keskivertorunoilijan päässä olevan ohjelmanhan on kirjoittanut runoilijan oma kulttuuri, ja tämän kulttuurin puolestaan on ohjelmoinut sitä edeltänyt kulttuuri ja niin edelleen aina aikojen aamuun asti, jolloin ne tiedonsirut, jotka myöhemmin osoittautuvat tärkeiksi tulevaisuuden runoilijalle, pyörteilivät vielä kosmoksen syövereiden alkukaaoksessa. Jotta siis voisi onnistuneesti ohjelmoida runokoneen, on ensin toistettava koko maailmankaikkeuden kehitys alusta pitäen — tai ainakin melkoinen osa siitä.

– Trurlin elektrubaduuri [Lem, 1965]

1.1 Ohjelmistojen tekijöiden ongelmakenttä

Tehokkaiden, luotettavien, halpojen, selkeiden, helppokäyttöisten,

ylläpidettävien, siirrettävien, pitkäikäisten — sanalla sanoen **laadukkaiden** ohjelmistojen tekijät ovat ammattilaisia, jotka joutuvat päivittäisessä työssään toimimaan taiteen ja tieteen rajamailla. Toisaalta ohjelmoijien on ymmärrettävä alansa formaalit periaatteet (ohjelmistojen suunnittelumenetelmistä tietorakenteiden käyttöön ja ohjelmointiin liittyvään matematiikkaan). Toisaalta ohjelmoijat ovat myös taiteilijoita, jotka muokkaavat olemassa olevista materiaaleista ja valmiista rakennuspalikoista ohjelmiston kokonaisuuden. Ohjelmien teon luova puoli on varsinkin alan palkkauksessa vähälle huomiolle jätetty puoli. Silti kaikki tuntevat alalla toimivia "taikureita", jotka hallitsevat bittejä ja niiden kokonaisuuksia muita paremmin.

1.2 Laajojen ohjelmistojen teon vaikeus

Jokainen tietokoneohjelmointia opiskellut on kirjoittanut ensimmäisenä ohjelmanaan korkeintaan muutaman kymmenen rivin ohjelman, jonka avulla on saanut tuntuman ohjelmoinnin käytännön puoleen (ohjelman kirjoittaminen syntaksisesti oikein koneen ymmärtämään muotoon, ohjelman suorittaminen tietokoneella ja mahdollisesti jonkin näkyvän tuloksen saaminen aikaan, ohjelman toimintalogiikan muuttaminen ja sovittaminen halutun ongelman ratkaisemiseksi jne.). Koska lyhyet ohjelmanpätkät saa useimmiten tekemään haluamiaan asioita kohtuullisen lyhyellä vaivannäöllä (muutamasta minuutista muutamaan päivään), on yleinen harhaluulo, että tästä voidaan yleistää suurempien ohjelmistojen valmistamisen olevan *vain* saman prosessin toistaminen isommalle rivimäärälle ohjelmakoodia.

Käytännössä jokaiselle ihmiselle tulee jossain vaiheessa vastaan raja tiedon hallinnassa, kun hallittava tietomäärä kasvaa liian suureksi. Ohjelmien teossa tämä raja tulee näkyviin ohjelman rivimäärän kasvaessa (muuttujista ei muista enää heti kaikkien käyttötarkoitusta, ehto- ja silmukkalauseiden logiikka hämärtyy, tietorakenteiden hallinta sekoaa, osoittimet eivät osoita oikeaan paikkaan jne.).

Ohjelmistotekniikan suurimpia ongelmia on suurten ohjelmistojen tekemisen vaikeus. Tämä ns. **ohjelmistokriisi** ("*software crisis*") on tavallaan tietotekniikan itsensä aiheuttama ongelma. Tietokonelaitteistojen mm. tallennus- ja prosessointitekniikan räjähdysmäinen kehitys on mahdollistanut jatkuvasti aikaisempaa laajempien ja mutkikkaampien ohjelmistojen suorittamisen, mutta näiden ohjelmisto-

jen tekemiseen tarvittavat menetelmät ja työkalut eivät ole kehittyneet läheskään yhtä nopeasti. Ohjelmistotuotannon professori *Ilkka Haikala* on sanonut aiheesta:

> “Ohjelmistokriisiä ratkomaan on kehitetty mm. uusia työkaluja ja työmenetelmiä. Tästä huolimatta ohjelmistotyön tuottavuuden kasvu on tilastojen mukaan ollut vuosittain vain noin neljän prosentin luokkaa. Muutamien vuosien välein kaksinkertaistuvaan ohjelmistojen keskimääräiseen kokoon verrattuna kasvu on huolestuttavan pieni.” [Haikala ja Märijärvi, 2002]

Ongelma ei ole mikään uusi ilmiö eikä ole kyse siitä, että se olisi havaittu vasta viime aikoina. Vuonna 1972 vastaanottaessaan Turing Award -palkintoa *Edsger W. Dijkstra* puhui aiheesta:

> “As long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem and now that we have gigantic computers, programming has become an equally gigantic problem. In this sense the electronic industry has not solved a single problem, it has only created them — it has created the problem of using its product.” [Dijkstra, 1972]

Suurten ohjelmistokokonaisuuksien hallintaan on kehitetty useita erilaisia menetelmiä, mutta mikään niistä ei ole osoittautunut muita selvästi paremmaksi viisasten kiveksi, joka ratkaisee kaikki ohjelmistotyön ongelmat. Näistä eri menetelmistä eniten huomiota ovat viime aikoina saaneet oliokeskeiset menetelmät, joissa nimen mukaisesti keskitytään olioihin. Mitä nämä oliot sitten ovat ja miten ne vaikuttavat ohjelmien suunnitteluun ja toteuttamiseen? Näihin kysymyksiin haemme vastausta seuraavissa luvuissa.

1.3 Suurten ohjelmistokokonaisuuksien hallinta

Seitsemänkymmentäluvulla ryhdyttiin kehittämään menetelmiä ohjelmistokriisin ratkaisemiseksi — tämä työ jatkuu yhä edelleen. Yksi

varhaisimmista ratkaisuperiaatteista on tuttu kaikesta inhimillisestä toiminnasta: ***ongelman jakaminen yhden ihmisen hallittaviin paloihin ja yksinkertaistaminen abstrahoimalla***. Tämä periaate on nähtävissä, kun tarkastellaan ohjelmoijien tärkeimpien työkalujen, ohjelmointikielten ja niiden periaatteiden kehitystä.

Mitä abstrahointi tarkoittaa? Voimme aloittaa vaikka sivistyssanakirjan selvityksellä:

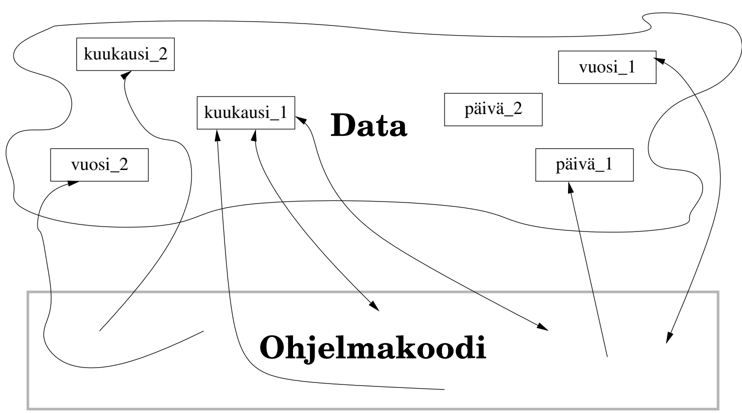
> **Abstraktio**. Ajatustoiminta, jonka avulla jostakin käsitteestä saadaan yleisempi käsite vähentämällä siitä tiettyjä ominaisuuksia. Myös valikointi, jossa jokin ominaisuus tai ominaisuusryhmä erotetaan muista yhteyksistään tarkastelun kohteeksi.
>
> **Abstrahoida**. Suorittaa abstraktio, erottaa mielessään olennainen muusta yhteydestä.
>
> **Abstrakti**. Abstrahoimalla saatu, puhtaasti ajatuksellinen, käsitteellinen. [Aikio ja Vornanen, 1992]

Kerätään siis yhteen toisiinsa liittyviä asioita ja nimitetään niiden kokonaisuutta jollain kuvaavalla “uudella” termillä tai kuvauksella. Ohjelmointi on täynnä tällaisia rakenteita. Otetaan esimerkiksi hyvin yleinen operaatio: tiedon tallentaminen massamuistilaitteelle. Sovellusohjelmoijan ei tarvitse tuntea kiintolevyn valmistajan käyttämää laitteen ohjauksen protokollaa, koodausta, ajoitusta ja bittien järjestystä, vaan hän voi käyttää erikseen määriteltyjä korkeamman (abstraktio)tason operaatioita (esim. `open`, `write` ja `close`).

Seuraavissa aliluvuissa näemme, miten abstrahoinnilla saadaan selkeämmäksi tietokoneohjelman rakenteen hallinta.

1.3.1 Ennen kehittyneitä ohjelmointikieliä: ei rakennetta

Ohjelmoinnin historian alkuhämärässä ohjelmoinnin tavan määräsi laitteisto. Prosessorien ymmärtämiä käskykoodeja kirjoitettiin suoraan joko niiden numerokoodeilla tai myöhemmin symboleja näiksi koodeiksi muuntavan assembler-ohjelman avulla. Tällaisessa laitteistonläheisessä näperryksessä kaikista hiemankin suuremmista ohjelmista tuli spagettiröykkiöitä, joissa tieto oli yksittäisiä muistipaikkoja, ja tietoa käsittelevää ohjelmakoodia oli ripoteltuna ympäriinsä. Kuvassa 1.1 seuraavalla sivulla on esimerkki ohjelmasta (koodi ja



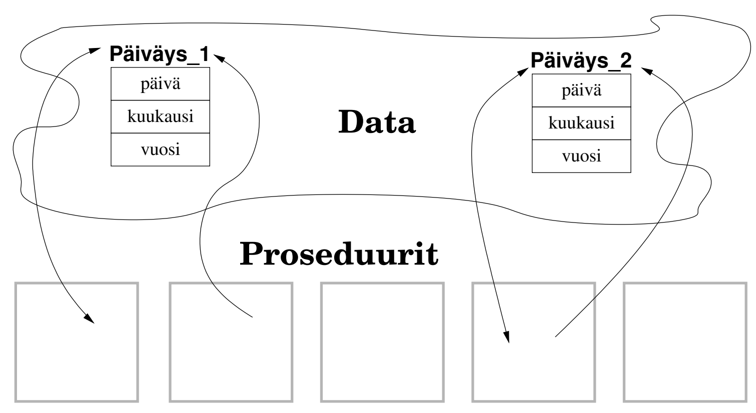
KUVA 1.1: Tyypillinen assembly-ohjelman spagettirakenne

data), joka käsittelee kahta päiväystä: ohjelmoija on varannut kolme muistipaikkaa yhdelle päiväykselle ja näitä tietoja käsitellään ohjelmakoodista suoraan muistipaikkojen osoitteilla. Rakennekuvasta näkee hyvin, että tällaisesta rakenteesta on esim. ylläpitovaiheessa hyvin vaikea selvittää, mitkä kaikki ohjelmakoodin kohdat viittaavat johonkin määrättyyn muistipaikkaan (esim. `kuukausi_1`).

1.3.2 Tietorakenteet: tiedon kerääminen yhteen

Kun ohjelmointikielten kehitys kuusikymmentäluvulla pääsi laajemmassa mitassa alkamaan, ryhdyttiin tekemään ohjelmoijien kokemusten pohjalta kieliä, joissa tietoa pystyttiin käsittelemään muistipaikkoja korkeammalla abstraktiotasolla — **tietorakenteina**.

Tietorakenteet ovat nimettyjä kokonaisuuksia, jotka koostuvat muista tietorakenteista tai ns. kielen perustyypeistä. Näiden hierarkkisten rakenteiden avulla saatiin kerättyä yhteen kuuluvat tiedot saman nimikkeen alle. Esim. päiväystieto voitaisiin haluta käsitellä kolmena kokonaislukuna, jotka kuvaavat vuotta, kuukautta ja päivää. Näistä voidaan muodostaa ohjelmointikielen tasolla yksi päiväykseksi nimetty tietorakenne. Vastaavasti ohjelman toiminnallisuutta voi-



KUVA 1.2: Päiväykset tietorakenteina

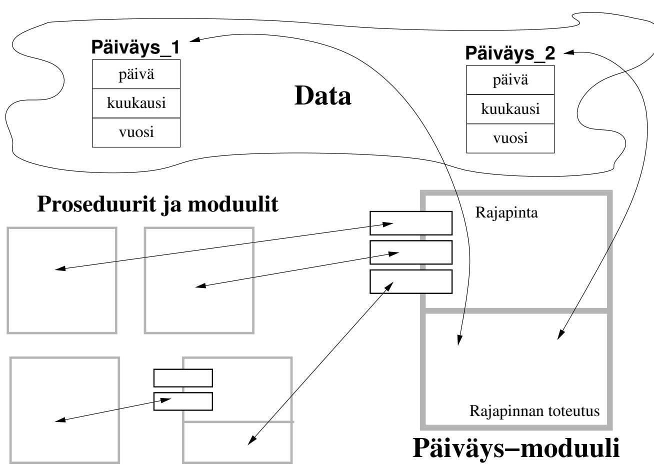
daan jakaa yhden toiminnon toteuttaviin paloihin, joita nimitetään funktioiksi tai proseduureiksi.

Kuvassa 1.2 on esimerkki kahdesta päiväyksestä ja niiden tietorakenteista. Tässä mallissa päiväyksiä käsitellään ohjelmakoodissa edelleen missä kohtaa tahansa aivan kuten spagettimallissakin. Tietorakenteita ja funktioita tukevia ohjelmointikieliä ovat mm. Pascal ja C.

1.3.3 Moduulit: myös ohjelmakoodi kerätään yhteen

Vuosi 2000 -ongelma oli tietotekniikassa paljon puhetta ja työtä synnyttänyt aihe. Siinä ohjelmistoista oli tarkastettava, että ne eivät oleta kaikkien vuosilukujen olevan muotoa $19xy$. Näiden kohtien löytäminen on edellisten käytäntöjen mukaan tehdyissä ohjelmistoissa hyvin vaikeata, sillä mikä tahansa ohjelmiston osa voi suoraan käsitellä päiväyksen sitä kokonaislukua, joka kuvaa vuosilukua.

Jotta tällaiset toiminnalliset viittaukset tietorakenteisiin saataisiin hierarkkiseen hallintaan, päätettiin määritellä joukko määrättyyn tietorakenteeseen liittyviä funktioita ja sopia, että ***vain näillä funktioilla saa käsitellä kyseistä tietorakennetta*** (esim. päiväys, katso kuva 1.3 seuraavalla sivulla). Tätä funktiokokoelmaa nimitetään **rajapinnaksi** (*interface*) ja sen funktioita **rajapintafunktioiksi**.



KUVA 1.3: Päiväyksiä käsittelevän moduulin rakenne

Ohjelmointikielten rakenteita, joissa kootaan tietorakenteet ja niitä käsittelevä ohjelmakoodi samaan kokonaisuuteen sekä erikseen määritellään tietorakenteiden käsittelyyn toiminnallinen rajapinta, nimitetään **moduuleiksi** (*module*). Koska moduulit kätkevät niiden rajapinnan toiminnallisuuden toteutuksen, moduuleja tulee tarkastella kahdesta näkökulmasta:

- **Moduulin käyttäjä.** Moduulin käyttäjä on ohjelmoija, joka tarvitsee moduulin tarjoamaa palvelua oman koodinsa osana. Tässä ulkopuolisessa näkökulmassa meitä kiinnostaa vain moduulin ulkoinen eli **julkinen rajapinta**: miten sitä tulee käyttää ja saammeko sen avulla toteutettua haluamamme toiminnan? (Moduulilla voi olla myös ns. sisäinen rajapinta, joka on tarkoitettu vain moduulin tekijän käyttöön.)

- **Moduulin toteuttaja.** Moduulin suunnittelijan ja toteuttajan vastuulla on määritellä ja dokumentoida moduulille rajapinta, joka on yksinkertainen, helppokäyttöinen, selkeä ja käyttökelpoinen. Vain moduulin toteuttajan täytyy miettiä ja toteuttaa rajapinnan takana oleva toiminnallisuus — tämä osa on muilta ohjelmoijilta kätkettynä siinä mielessä, että heidän ei vält-

tämättä tarvitse tuntea toteutuksen yksityiskohtia. Tätä nimitetään **tiedon kätkennäksi**. Siinä **kapseloidaan** käytön kannalta turha tieto moduulin sisälle (*encapsulation*). Kyseessä on abstrahoinnin tärkeimpiä työkaluja ohjelmoinnissa.

Käytännössä moduuliajattelu voi olla vain sopimus määrätyn rajapinnan noudattamisesta (esim. Pascal ja C), tai ohjelmointikieli voi jopa kieltää (ja tarkastaa) rajapinnan takana olevien tietorakenteiden käsittelyn moduulin ohjelmakoodin ulkopuolelta (Ada, Modula).

Rajapinta-ajattelu pakottaa suunnittelemaan tarkemmin ennalta, miten määrättyä tietorakennetta on tarkoitus käyttää eli mitä palveluita tietorakenteen lisäksi halutaan siihen liittyen tarjota. Tiedon kätkennän ja rajapinta-ajattelun haittapuolina voidaankin pitää suunnittelun vaikeutumista. Moduulin tekijälle on helppo sanoa päämääräksi yksinkertainen ja käyttökelpoinen (eli täydellinen?) rajapinta. Näiden vaatimusten toteuttamiseen sitten tarvitaankin todellista ohjelmointitaikuria. Käytännön ohjelmistotyössä rajapintoja joudutaan tarkastamaan ja tarkentamaan ohjelmiston kokonaissuunnittelun edetessä ja jatkossa ohjelmistoa ylläpidettäessä. [Sethi, 1996]

Rajapinta edistää merkittävästi ohjelmistojen ylläpidettävyyttä, koska ne rajapinnan "takana" tehdyt muutokset, jotka eivät vaikuta rajapinnalle määriteltyyn toiminnallisuuteen, eivät aiheuta mitään muutoksia muuhun ohjelmistoon. Näin saadaan abstrahoitua laajan ohjelmiston kokonaisuutta rajapinnoilla toisistaan eroteltuihin mahdollisimman itsenäisiin paloihin.

Vuosi 2000 -ongelman tapauksessa modulaarisessa ohjelmistossa on ensin tarkastettava, että rajapinnan kautta ei päästä käsittelemään vuosilukuja väärällä tavalla. Jos vuosisatojen 1900 ja 2000 väliseen käsittelyyn liittyvät rakenteet ovat mahdollisia vain moduulin sisällä, niin riittää, että tarkastamme kaiken moduulin koodin ja varmistamme sen toimivan myös vuoden 1999 jälkeen. On tärkeätä huomata, että jos rajapinnan määrittely "paljastaa" vuosiluvun tietorakenteen ulkopuolelle esimerkiksi 0–99 välille määriteltynä kokonaislukuna, niin joudumme edelleen tarkastamaan myös kaiken moduulin ulkopuolella päiväyksiä käsittelevän ohjelmakoodin. Tällaista rajapintaa voidaan pitää huonosti suunniteltuna vuosi 2000 -ongelman kannalta. Kyse on kuitenkin jälkiviisaudesta, jos rajapinnan suunnittelun aikaan ohjelmiston arvioitu elinkaari ei ole yltänyt vuoden 1999 yli- se. Olisiko rajapinnan määrittelijän pitänyt ottaa huomioon ongelma-

kentän ulkopuolisia asioita? Ja mitä kaikkia niistä? Rajapinta-ajattelu ei ole inhimillisiä virheitä korjaava tai estävä menetelmä, mutta se on ihmisille luontaista hierarkkista ajattelua tukeva menetelmä, joka selkeyttää suurten ohjelmistojen rakennetta.

Modulaarista ohjelmointia tukevia ohjelmointikieliä ovat mm. Ada ja Modula. Listauksessa 1.1 on esimerkki päiväys-moduulin tietorakenteesta ja osasta rajapintaa Modula-3-kielellä.

1.3.4 Oliot: tietorakenteet, rajapinnat ja toiminnot yhdessä

Kun tarkastelemme päiväysrajapinnan käyttöä (listaus 1.2 seuraavalla sivulla), niin huomaamme heti, että rajapintafunktioille on aina välitettävä parametrina se tietorakenne, johon operaatio halutaan kohdistaa. Tästä käytännön tarpeesta on lähtöisin ajattelumalli, jossa ei haluta korostaa pelkästään moduulin toiminnallista osaa (rajapintafunktiot) vaan ajatellaan rajapintaa tietorakenteen "osana". Tämän mallin mukaisia alkioita, jotka yhdistävät tietorakenteet ja rajapinnan, nimitetään olioiksi (kuva 1.4 seuraavalla sivulla). Uudessa ajattelumallissa kohdistetaan rajapintakutsu määrättyyn olioon, ja tämä näkyy myös olio-ohjelmointikielen tasolla (listaus 1.3 sivulla 36).

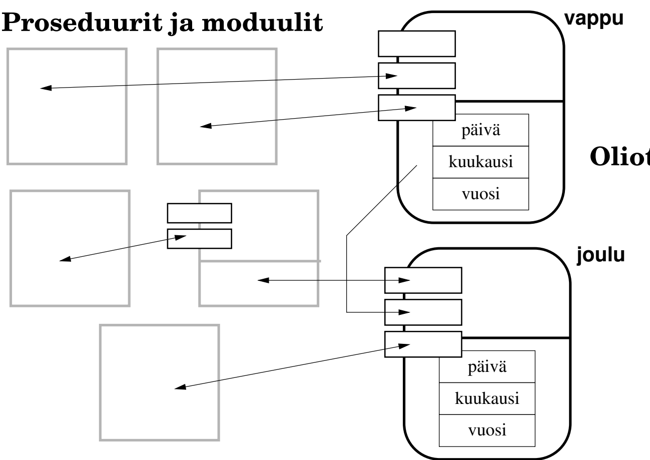
Oliomallin mukaisen ohjelmistojen suunnittelun ja toteutuksen katsotaan sopivan paremmin ihmisten luontaiseen tapaan tarkastella ongelmia. Oli toteutettavana järjestelmänä lentokoneen toiminnan valvonta tai šakkipeli, niin järjestelmän katsotaan koostuvan osista ja näiden osien toisiinsa ja ulkomaailmaan kohdistamista toiminnoista.

```
1  INTERFACE päiväys;
2
3  TYPE
4    PVM : RECORD
5      päivä, kuukausi, vuosi : INTEGER;
6    END;
7
8  PROCEDURE Luo( päivä, kuukausi, vuosi : INTEGER ) : PVM;
9  PROCEDURE PaljonkoEdellä( eka, toka : PVM ) : INTEGER;
10 PROCEDURE Tulosta( kohde : PVM );
```

LISTAUS 1.1: Päiväysmoduulin tietorakenne ja osa rajapintaa

```
1 IMPORT päiväys;
2
3 VAR vappu : päiväys.PVM;
4
5 BEGIN
6   vappu := päiväys.Luo( 1, 5, 2001 );
7   päiväys.Tulosta( vappu );
8 END;
```

LISTAUS 1.2: Päiväysmoduulin käyttöesimerkki



KUVA 1.4: Rajapinta osana tietorakennetta (oliot)

Tämä uusi ajattelumalli vaikuttaa useisiin ohjelmistojen tekemisen osa-alueisiin:

- **Määrittely.** Käytettäessä olioita on määrittelyssä pidettävä mielessä tämä päämäärä (päädytään oliorakenteeseen).
- **Suunnittelu.** Oliosuunnittelussa on tunnettava olioiden ja luokkien ominaisuuksia (mm. periytyminen ja geneerisyys), jotta jo suunnitteluvaiheessa pystytään hyödyntämään uuden ajattelumallin kaikki (tarvittava) teho.

```
1  PROCEDURE TulostaAikaisempi( päivä_A, päivä_B : PVM )
2  BEGIN
3    IF päivä_A.PaljonkoEdellä( päivä_B ) > 0 THEN
4      päivä_A.Tulosta();
5    ELSE
6      päivä_B.Tulosta();
7    END;
8  END;
9
10 BEGIN
11   vappu := PVM( 1, 5, 2000 );
12   joulu := PVM( 24, 12, 1999 );
13   TulostaAikaisempi( vappu, joulu );
14 END.
```

LISTAUS 1.3: Päiväysolioiden käyttö

- **Ohjelmointi.** Erityiset olio-ohjelmointikielet helpottavat olioita sisältävän suunnitelman toteuttamista huomattavasti. Työkaluina nämä kielet ovat aikaisempia mutkikkaampia, koska ne tuovat lisää uusia laajoja ominaisuuksia aikaisempiin (proseduraalisiin) kieliin verrattuna.

Perinteisessä ohjelmistosuunnittelussa on otettu lähtökohdaksi osat (data, tietorakenteet) *tai* toiminnot (funktiot ja rajapinnat) ja pyritty rakentamaan koko ohjelmisto tästä yhdestä jaottelusta lähtien. Olio-ohjelmoinnin yksi tärkeimmistä suunnitteluperiaatteista on jatkuvasti yrittää pitää näitä molempia näkökantoja mukana suunnitteluprosessissa: järjestelmästä tunnistetaan sekä olioita että niiden välisiä toiminnallisuuksia.

Ongelmalähtöinen ja laitteistoläheinen näkökulma olioihin

Olion voidaan katsoa olevan osa sen ongelman ratkaisua, johon ohjelmistolla pyritään. Tämä on ulkoinen tai ongelmalähtöinen näkökulma oliohin. Ohjelmiston suunnittelija pyrkii tunnistamaan esim. šakkipelistä pelin mallintamiseen tarvittavat oliot: erilaiset nappulat, lauta, peliajan mittaava kello jne.

Olioista koostuvan ohjelmiston toiminnallisuus on olioiden välistä kommunikointia. Oliot kutsuvat toistensa rajapintojen kautta tarvitsemiaan toimintoja. Esimerkiksi šakkinappulaolio voi pyytää šak-

kilautaa mallintavalta oliolta tietoa siitä, voiko se siirtyä määrättyyn ruutuun. Kyselyn seurauksena lautaolio voi taas vuorostaan kysyä muilta nappuloilta niiden nykyisiä paikkoja jne. Ohjelmiston ylimmän tason "logiikka" voidaan näin määritellä olioiden rajapintojen ja olioiden välisen kommunikaation avulla — tässä suunnitteluvaiheessa ei tarvitse ottaa kantaa siihen, miten nämä toiminnot tullaan yksityiskohtaisesti toteuttamaan.

Toisaalta jokaisen olion voi ajatella edustavan erillistä tietokonetta, jolla on oma ohjelmakoodi (rajapinnan takainen toteutus) ja muisti (olion tietorakenteen arvot eli olion tila). Nämä "minikoneet" kommunikoivat lähettämällä toisilleen viestejä, joilla pyydetään jonkin toiminnon suorittamista toisessa koneessa eli oliossa. Tämä ajattelumalli on sisäinen tai laitteistonläheinen näkökulma olioihin. Malli voi olla hyödyllinen esimerkiksi hajautetuissa järjestelmissä, joissa osa ohjelmiston olioista tulee lopullisessa toteutuksessa todellisuudessakin sijaitsemaan eri fyysisissä tietokoneissa. [Sethi, 1996]

Ohjelmiston staattiset ja dynaamiset osat

Oliomallia ei ole tarkoitettu syrjäyttämään modulaarisuutta vaan täydentämään sitä. Moduulien avulla ohjelmistoon voidaan tehdä yleensä ylimmillä tasoilla olevaa jakoa osiin ja niiden välisiin rajapintoihin: käyttöliittymän näkymät, näyttölaitteiden rajapinnat, tietokanta jne. Koska tämä jaottelu osiin on ohjelmassa pysyvä, sitä nimitetään staattiseksi jaotteluksi.

Dynaamiset osat tarkoittavat tietorakenteiden ja rajapintojen kokoelmia, joissa yhden mallin mukaisia olioita voi esiintyä ohjelman ajoaikana useita. Esim. päiväysoliolla on aina sama toiminnallinen rajapinta ja sama tietorakenne (kolme kokonaislukua). Eri päiväyksillä voi olla erilaiset arvot tietorakenteessa. Saman mallin (päiväys) eri ilmentymillä (oliot) sanotaan olevan erilainen tila (tietorakenteen arvot). Olioiden dynaamisuutta on myös se, että niitä voi syntyä ja tuhoutua ohjelman suorituksen aikana — kutakin moduulia taas on olemassa yksi kappale koko ohjelmiston ajossaolon ajan.

Olio on itsenäinen kokonaisuus

Oliokeskeisessä ajattelumallissa olioiden katsotaan olevan itsenäisiä kokonaisuuksia, joille on määritelty tarkka **vastuualue** (*responsibil-*

ity) ohjelmistossa [Budd, 2002]. Vastuualue on nimitys kaikelle sille, mitä olion on määrätty toteuttavan. Esim. päiväysoliolla on määriteltynä vastuu tiedon taltioinnista (olio kuvaa yhtä päivämäärää) ja julkinen rajapinta (tarjotut palvelut). Vastuualue kattaa muutakin kuin ohjelmointikielen muuttujia ja funktioita. Se voi esimerkiksi määritellä että jokin olio on vastuussa määrättyjen muiden olioiden luomisesta ja tuhoamisesta eli niiden elinkaaresta.

1.3.5 Komponentit: moduulit ja oliot yhdessä

On tärkeätä huomata, että moduulit ja oliot voivat muodostaa hierarkioita ja kokonaisuuksia yhdessä. Ohjelmistossa voidaan esim. ylimmällä tasolla määritellä ajanlaskusta vastuussa oleva moduuli, jonka tarjoamista palveluista yksi on päiväyksiä kuvaavat oliot.

Hyvin suunniteltu rajapintojen, olioiden ja toteutusten (samalla rajapinnalla voi olla esimerkiksi vaihtoehtoisia toteutuksia) kokoelmaa on viime aikoina ryhdytty nimittämään **komponentiksi** (*component*). Ideana olisi tarjota ohjelmistojen valmistajille hieman samanlainen tilanne kuin tällä hetkellä on mm. elektroniikkasuunnittelijoilla — he voivat suunnitella ja valmistaa tuotteitaan hyvin laajan valmiin komponenttivalikoiman avulla. Tuote luodaan yhdistelemällä elektroniikkakomponentit uusiksi mutkikkaammiksi tuotteiksi. Vastaavalla tavalla uudelleenkäyttöä varten suunniteltuja ohjelmakomponentteja voitaisiin pitää uusien ohjelmistojen pohjana.

Ohjelmistokomponentti on itsenäinen kokonaisuus, jota sen tarjoamien julkisten rajapintojen avulla voidaan hyödyntää osana suurempaa kokonaisuutta. Komponentti kerää yhteen määrätyn vastuualueen toteuttamiseen tarvittavat ohjelmiston moduulit ja oliot (olioohjelmana toteutus on usein kokoelma luokkia). Komponenttimarkkinoita on tällä hetkellä olemassa erityisesti graafisten käyttöliittymien alueella, mutta vasta tulevaisuus tulee näyttämään, onko kyseessä ohjelmistojen tekemistä suuremmassa mittakaavassa muokkaava ajattelumalli.

Aivan viime vuosien ohjelmistokomponentit sisältävät usein rajapintadokumentaation ja lähdekoodin lisäksi myös valmiin binäärimuotoisen toteutuksen. Tällöin komponentti otetaan sellaisenaan (valmiiksi konekoodiksi käännettynä) käyttöön omaan ohjelmistoon. Yksi tällainen komponenttiarkkitehtuuri on JavaBeans [JavaBeans, 2001].

1.4 C++: Moduulit käännösyksiköillä

Seuraavassa luvussa esiteltävä C++:n luokkarakenne sisältää olio-ominaisuuksien lisäksi modulaarisuuden tärkeimmät piirteet: julkisen rajapinnan ja toteutuksen kätkennän. Tässä ja seuraavassa aliluvussa esitellään kaksi tapaa tehdä staattisia moduulirakenteita C++:lla.

Staattisella moduulilla tarkoitetaan käännösaikana luotua ja koko ohjelman suoritusajan samana pysyvää kokonaisuutta, jonka myös kääntäjä ymmärtää erilliseksi yksiköksi. C ja C++ -kääntäjät ovat aina käsitelleet yhtä lähdekooditiedostoa kerrallaan yhtenä kokonaisuutena, jonka käännöksessä pitää olla näkyvillä kaikkien käytettyjen rakenteiden esittelyt. [Kerninghan ja Ritchie, 1988]

Ohjelmoitavan moduulin julkisen rajapinnan esittely voidaan laittaa ns. **otsikkotiedostoon** (*header*, listaus 1.4), joka taas voidaan **#include**-esikääntäjäkomennolla ottaa näkyville kaikkiin moduulia käyttäviin tiedostoihin.

Suurissa ohjelmissa **#include**-rakenteet tulevat mutkikkaiksi ja monitasoisiksi. Kääntäjä voi tällöin virheellisesti saada käsittelyyn saman otsikkotiedoston useaan kertaan saman käännöksen aikana, mikä on virhetilanne koska kieli sallii esittelyiden esiintyvän vain kerran samassa käännöksessä. Esimerkiksi päiväysmoduulin esittelyä tarvitaan useassa korkeamman tason moduulin otsikkotiedostossa, jotka käännösyksikkö ottaa näkyville **#include**-käskyllä, kuten kuva 1.5 seuraavalla sivulla näyttää. Esimerkkilistauksen 1.4 alussa näkyvällä esikääntäjän ehdollisella kääntämisellä (**#ifndef** X **#define** X

```
1  #ifndef PAIVAYS_H
2  #define PAIVAYS_H
3
4  typedef struct paivays_data {
5    int p_, k_, v_;
6  } paivays_PVM;
7
8  paivays_PVM paivays_luo( int paiva, int kuukausi, int vuosi );
9  void paivays_tulosta( paivays_PVM kohde );
         ⋮
10 #endif /* PAIVAYS_H */
```

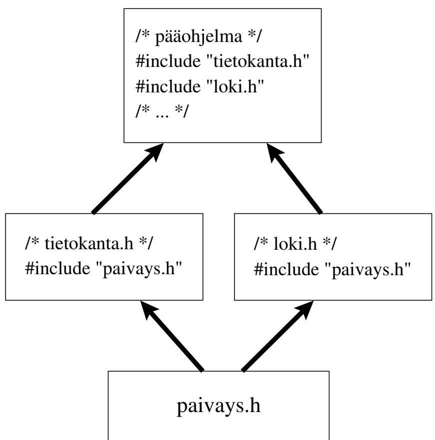
LISTAUS 1.4: Päiväysmoduulin esittely C-kielellä, `paivays.h`

... **#endif**) saadaan varmistettua, että esittely näkyy kääntäjälle vain yhden kerran.

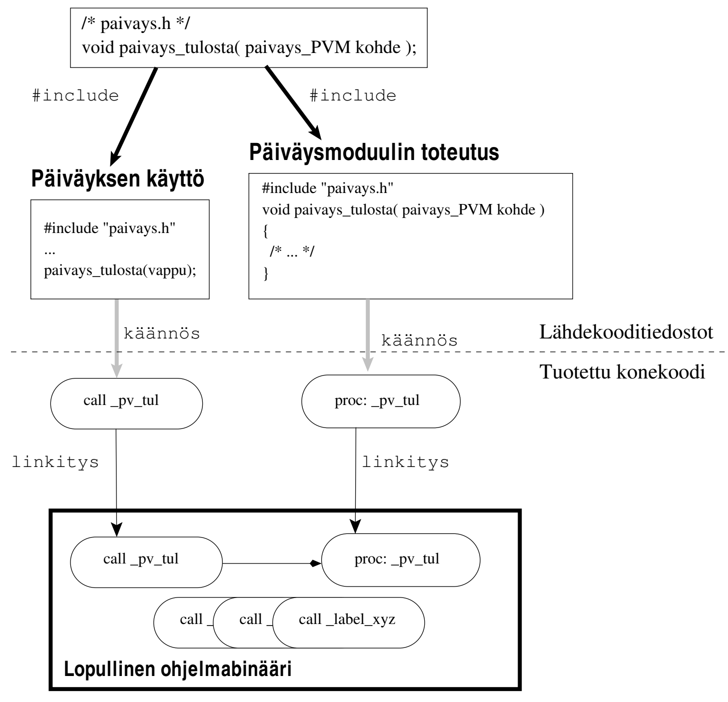
Eri puolella ohjelmistoa (kooditiedostoja) esitellyt rakenteet toteutetaan yhdessä käännösyksikössä, joka lopuksi linkitetään mukaan valmiiseen ohjelmaan. Käännösyksikkö on C-kielessä yksi tiedosto (joka mahdollisesti on ottanut käännökseen mukaan toisia tiedostoja **#include**-rakenteella), josta kääntäjä tekee objektitiedoston. Linkitysvaihe pitää huolen siitä, että esittelyosan nähneiden käännösyksiköiden kutsut moduulin funktioihin menevät oikeaan osaa ohjelmaa lopullisessa suoritettavassa ohjelmabinäärissä (kuva 1.6 seuraavalla sivulla). Tässä ratkaisussa on kaksi ongelmaa:

1. Kaikki eri moduulien käyttämät nimet (funktioiden ja muuttujien nimet) ovat oletuksena käytettävissä kaikkialla ohjelmassa (kunhan ne esitellään käännösyksikössä). Käännösyksikön paikalliseen käyttöön tarkoitetut nimet on ohjelmoijan itse merkittävä määreellä **static**.

2. On olemassa vain yksi ylimmän tason (globaali) näkyvyysalue, jossa kaikki moduulien julkisten rajapintojen symbolit ovat näkyvissä. Jos esim. usea moduuli esittelee funktion `Tulosta`, niin



KUVA 1.5: Sama otsikkotiedosto voi tulla käyttöön useita kertoja



KUVA 1.6: Moduulirakenne C-kielellä

ohjelma ei käänny, koska sama symboli on lopullisessa binäärissä määriteltynä useita kertoja. Tämän nimiavaruuden "roskaantumisen" takia on moduulin toteuttajan pyrittävä nimeämisellä välttämään nimikonflikteja. Esimerkkissä tämä on tehty liittämällä moduulin nimiin etuliite "`paivays_`".

1.5 C++: Nimiavaruudet

Ratkaisuksi suurten ohjelmistojen rajapintojen nimikonflikteihin ja modulaarisen rakenteen esittämiseen ISOC++ -standardi [ISO, 1998] tarjoaa **nimiavaruudet** (*namespace*). Nimiavaruuksien tarkoituksena

on tarjota kielen syntaksin tasolla oleva hierarkkinen nimeämiskäytäntö. Hierarkia auttaa jakamaan ohjelmistoa osiin sekä käytännön ohjelmoinnissa estää nimikonflikteja ohjelmiston eri moduulien välillä. Kuvaavat ja tunnetut nimet voivat esiintyä suuressa ohjelmistossa useassa paikassa ja ne täytyy pystyä erottamaan toisistaan. Esimerkiksi aliohjelmat `Päiväys::tulosta()` ja `Kirjastonkirja::tulosta()` suorittavat saman semanttisen operaation (tulostamisen), mutta etuliite kertoo, minkä moduulin määrittelemästä tulostusoperaatiosta on kyse.

1.5.1 Nimiavaruuksien määritteleminen

Käytännössä C++-ohjelmoija voi uudella avainsanalla **`namespace`** koota yhteen toisiinsa liittyviä ohjelmakokonaisuuksia, jotka voivat olla mitä tahansa C++-ohjelmakoodia. Kaikki esimerkkimme päivämääriä kuvaavat ohjelmiston osat (tietorakenteet, tietotyypit, vakiot, oliot ja funktiot) voidaan kerätä yhteen nimikkeen **`namespace`** `Paivays`[ϒ] alle, kuten listauksessa 1.5 seuraavalla sivulla on tehty.

Nimiavaruuden voi toisessa käännösyksikössä (eli lähdekooditiedostossa) “avata” uudelleen laajentamista varten, jolloin moduulin esittelemät rajapintafunktiot voidaan toteuttaa toisessa ohjelmatiedostossa (listauksessa 1.6 seuraavalla sivulla). Tällä tavoin moduulin esittely ja toteutus saadaan eroteltua toisistaan. Tiedon yksinkertaistamisen (abstrahointi) mukaisesti moduulin käyttäjän pitäisi pystyä hyödyntämään moduulia ainoastaan sen esittelyä (`hh`-tiedosto) ja dokumentointia tutkimalla. “Tylsät” ja epäoleelliset toteutusyksityiskohdat on piilotettu erilliseen käännösyksikköön, joka kuitenkin on osa samaa C++-nimiavaruutta.

1.5.2 Näkyvyystarkenninoperaattori

Kaikki nimiavaruuden sisällä olevat tunnistenimet ovat oletuksena näkyvissä eli käytettävissä vain kyseisen nimiavaruuden sisällä (`paivays.hh`-tiedostossa esitelty tietue `PVM` on käytössä listauksessa 1.6 seuraavalla sivulla). Ulkopuolelta nimiavaruuden sisällä ole-

[ϒ] ISOC++ -standardi sallisi tunnisteiden nimissä mm. skandinaavisia symboleja, jolloin esimerkkimme nimiavaruuden nimi olisi `Päiväys`. Koska käytännössä kääntäjät eivät tällaista ominaisuutta tue, ohjelmaesimerkkiemme nimet ovat ilman suomen kielen kirjaimia å, ä ja ö.

```
1  // Päiväys-moduulin rajapinnan esittely (tiedosto: paivays.hh)
2  #ifndef PAIVAYS_HH
3  #define PAIVAYS_HH
4
5  namespace Paivays {
6
7      // Päiväyksien tietorakenne:
8      struct Pvm { int p_, k_, v_; };
9
10     // Julkisen rajapinnan funktiot:
11
12     Pvm luo( int paiva, int kuukausi, int vuosi );
13             // Palauttaa uuden alustetun päiväyksen.
14
15     void tulosta( Pvm kohde );
16             // Tulostetaan päiväys järjestelmän oletuslaitteelle.
          ⋮
17 }
18 #endif
```

LISTAUS 1.5: Nimiavaruudella toteutettu rajapinta

```
1  // Päiväys-moduulin toteutus (tiedosto: paivays.cc)
2  #include "paivays.hh"        /* rajapinnan esittely */
3
4  namespace Paivays {
5
6      Pvm luo( int paiva, int kuukausi, int vuosi )
7      {
8          Pvm paluuarvo;
          ⋮
9          return paluuarvo;
10     }
11
12     void tulosta( Pvm p )
13     {
          ⋮
14     }
          ⋮
15 }
```

LISTAUS 1.6: Rajapintafunktioiden toteutus erillisessä tiedostossa

viin rakenteisiin voidaan viitata **näkyvyystarkenninoperaattorin** (`::`) avulla (listaus 1.7).

Tarkoituksena on, että ohjelmoija kertoo käyttöpaikassa mitä kokonaisuutta ja mitä alkiota sen sisällä hän kulloinkin tarkoittaa (onko käytössä funktio `Paivays::tulosta()` vai `Kirja::tulosta()`). Tapaa voidaan kritisoida ylimääräisellä kirjoitusvaivalla, mutta tässä kannattaa huomioida myös mukaan tuleva dokumentaatio. Moduulinimet kertovat heti, mitä rakenteita käytetään, eikä niitä luultavasti tarvitse enää erikseen kommentoida ohjelmaan.

Nimiavaruuksia voi olla myös sisäkkäin, jolloin näkyvyystarkentimia ketjutetaan oikean alkion osoittamiseksi (`Tietokanta::Yhteys::SQL`). C++ ei aseta mitään erityisiä rajoja tämän ketjun pituudelle, mutta eivät pitkät ketjut enää noudata nimiavaruuksien alkuperäistä ajatusta selkeydestä hierarkian avulla.

1.5.3 Nimiavaruuksien hyödyt

Toteuttamalla moduulit nimiavaruuksien avulla saadaan C-kielen malliin verrattuna seuraavat parannukset:

- Nimikonfliktien vaara vähenee merkittävästi, koska jokaisen moduulin rajapintanimet ovat omassa nimetyssä näkyvyysalueessaan (tietysti todella laajoissa ohjelmistoissa on pidettävä huoli siitä, etteivät nimiavaruuksien nimet taas vuorostaan ole samoja eri moduuleilla).

- Moduulin määrittelemien rakenteiden käyttö on kielen syntaksin tasolla näkyvän rakenteen (näkyvyystarkennin `::`) vuoksi

```
1 #include "paivays.hh"
2
3 int main() {
4   Paivays::Pvm vappu;
5   vappu = Paivays::luo( 1,5,2001 );
6   Paivays::tulosta( vappu );
        ⋮
7
```

LISTAUS 1.7: Nimiavaruuden rakenteiden käyttäminen

selkeämpää (vertaa esim. Modula-3:n moduuliproseduurien kutsutapa pisteoperaattorilla listauksessa 1.2 sivulla 35).

- Hierarkkisuudesta huolimatta moduulin sisällä on käytettävissä lyhyet nimet. Koodista nähdään syntaksin tasolla esimerkiksi, mitkä funktiokutsut kohdistuvat saman moduulin sisälle ja mitkä muualle ohjelmistoon.

Nimiavaruudet ovat kehittyneet C++:aan vähitellen korjaamaan havaittuja puutteita. Ne ovat melko uusi ominaisuus, joten on olemassa paljon C++ ohjelmakoodia, jossa niitä ei käytetä, mutta nimiavaruuksien etujen takia niiden käyttöä suositellaan yleisesti.

1.5.4 `std`-nimiavaruus

ISOC++ -standardi määrittelee omaan käyttöönsä `std`-nimisen nimiavaruuden. Tämän nimen alle on kerätty kaikki kielen määrittelyn esittelemien rakenteiden nimet (muutamaa poikkeusta, kuten funktiota `exit`, lukuun ottamatta). Esimerkiksi C:n tulostusrutiini `printf` ja C++:n tulostusolio `cout` ovat ISOC++:n mukaisesti toteutetussa kääntäjässä nimillä `std::printf` ja `std::cout`, jotta ne eivät aiheuttaisi ongelmia ohjelman muiden nimien kanssa. Koska kirjastoon kuuluu myös C-kielestä tulleita funktioita, `std`-nimiavaruus sisältää satoja nimiä. `std`-nimiavaruus on olemassa kaikissa nykyaikaisissa C++-kääntäjissä, ja ainoastaan sen rakenteiden käyttäminen on sallittua — tähän nimiavaruuteen ei saa itse lisätä uusia ohjelmarakenteita.

1.5.5 Standardin uudet otsikkotiedostot

Samalla kun kielen määrittelemät rakenteet on siirretty `std`-nimiavaruuden sisään, ovat myös otsikkotiedostojen nimet vaihtuneet. Aikana ennen nimiavaruuksia C++:ssa otettiin tulostusoperaatiot käyttöön esiprosessorin käskyllä `#include <iostream.h>`. Nyt oikea (`std`-nimiavaruudessa oleva) tulostusoperaatioiden esittely saadaan käskyllä `#include <iostream>`. Tämä tiedostotyyppiin viittava `.h`-liitteen puuttuminen on ominaisuus kaikissa standardin määrittelemissä otsikko-“tiedostoissa”. Sana “Tiedostot” on lainausmerkeissä, koska standardi ei määrää rakenteiden olevan missään tiedostossa, vaan em. rivi ainoastaan kertoo kääntäjälle, että kyseiset esittelyt otetaan

käyttöön — kääntäjä saa toteuttaa ominaisuuden vapaasti vaikka tietokantahakuna.

C-kielen kautta standardiin tulleiden otsikkotiedostojen nimistä on myös poistunut loppuliite “.h” ja lisäksi niiden eteen on lisätty kirjain “c” korostamaan C-kielestä peräisin olevia rakenteita. Esimerkiksi merkkitaulukoiden käsittelyyn tarkoitetut funktiot (strcpy yms.) saadaan käyttöön käskyllä **#include** <cstring>. Näiden “vanhojen” funktioiden käytöstä yhdessä C++-tulostusoperaatioiden kanssa näkyy esimerkki listauksessa 1.8.

1.5.6 Nimiavaruuden synonyymi

Nimiavaruuden nimi voi myös olla synonyymi (alias) toiselle jo olemassa olevalle nimiavaruudelle. Tällöin kaikki alias-nimeen tehdyt viittaukset käyttäytyvät alkuperäisen nimen tavoin. Tätä ominaisuutta voidaan hyödyntää esimerkiksi silloin, jos samalle moduulille on olemassa useita vaihtoehtoisia toteutuksia. Moduulia käyttävä ohjelmakoodi kirjoitetaan käyttämällä alias-nimeä ja todellinen käyttöön otettava moduuli valitaan synonyymin määrittelyn yhteydessä (katso listaus 1.9 seuraavalla sivulla).

Kun ollaan nimeämässä yleiskäyttöisiä ohjelmakoodikirjastoja, on tärkeätä valita myös kokonaisuuden nimiavaruudelle nimi, joka ei helposti aiheuta nimikonfliktia muun ohjelmiston kanssa. Esimer-

```
1  #include <cstdlib>      // pääohjelman paluuarvo EXIT_SUCCESS
2  #include <iostream>     // C++:n tulostus
3  #include <cstring>      // C:n merkkitaulukkofunktiot
4
5  int main()
6  {
7    char const* const p = "Jyrki Jokinen";
8    char puskuri[ 42 ];
9    std::strcpy( puskuri, "Jyke " );
10   std::strcat( puskuri, std::strstr(p, "Jokinen") );
11
12   std::cout << puskuri << std::endl;
13   return EXIT_SUCCESS;
14 }
```

LISTAUS 1.8: C-kirjastofunktiot C++:n nimiavaruudessa

```
1  #include "prjlib/string.hh"
2  #include <string>
3
4  int main() {
5  #ifdef PRJLIB_OPTIMOINNIT_KAYTOSSA
6    namespace Str = ComAcmeFastPrjlib;
7  #else
8    namespace Str = std;
9  #endif
10
11   Str::string esimerkkijono;
          ⋮
12 }
```

LISTAUS 1.9: Nimiavaruuden synonyymi (aliasointi)

kiksi `String` lienee huono kirjaston nimi, sillä usealla valmistajalla on varmasti kiinnostusta tehdä samanlainen rakenne. Virallisen nimen kannattaa olla pitkä (`OhjTutFiOpetusMerkkijono`), ja ohjelmoijat voivat omassa koodissaan käyttää moduulia lyhyemmällä etuliitteellä synonyymin avulla.

1.5.7 Lyhyiden nimien käyttö (`using`)

Etuliitteen "`std::`" toistaminen jatkuvasti esimerkiksi `cout`-tulostuksessa on varmasti raskaalta ja turhalta tuntuva rakenne.[♉] Jos samassa ohjelmalohkossa käytetään useita kertoja samaa nimiavaruuden sisällä olevaa nimeä, ohjelmoijien kirjoitusvaivaa helpottamaan on olemassa **`using`**-lause, joka "nostaa" nimiavaruuden sisällä olevan nimen käytettäväksi ohjelman nykyisen näkyvyysalueen sisälle. Peruskäytössä kerrotaan yksittäinen rakenne, jota halutaan käyttää. Esimerkiksi **`using`** `std::cout` mahdollistaa tulostusolion nimen käytön ilman etuliitettä (katso listaus 1.10 seuraavalla sivulla). Jotta nimiavaruuksien alkuperäinen käyttötarkoitus säilyisi, **`using`**-lauseet tulisi laittaa aina mahdollisimman lähelle niiden tarvittua käyttökohtaa ohjelmatiedostossa (funktion tai koodilohkon alkuun, jossa ne samalla toimivat dokumenttia käytetyistä ulkopuolisista rakenteista).

[♉] Kaikki C++-kääntäjät eivät vaadi `std::`-etuliitteen käyttämistä, mutta kyseessä on kielen standardin vaatimuksen vastainen toiminnallisuus.

```
1  #include "paivays.hh"
2  #include <iostream>
3
4  void kerroPaivays( Paivays::Pvm p )
5  {
6    using std::cout; // käytetään määrättyä nimeä
7    using std::endl;
8    using namespace Paivays; // käytetään kaikkia nimiavaruuden nimiä
9    cout << "Tänään on: ";
10   tulosta( p ); // Kutsuu rutiinia Paivays::Tulosta
11   cout << endl;
12 }
```

LISTAUS 1.10: using-lauseen käyttö

Turvallisuussyistä **using** tulisi laittaa ohjelmakooditiedoston alkuun vasta kaikkien #include-käskyjen jälkeen, jottei se vahingossa sotke otsikkotiedostojen toimintaa väärään paikkaan nostetulla nimellä. Sama sotkemisen välttäminen tarkoittaa myös sitä, ettei **using**-lauseita kannata laittaa otsikkotiedostojen sisälle (etukäteen ei tiedetä missä yhteydessä tai järjestyksessä tiedoston sisältämiä esittelyitä käytetään). [Sutter, 2000]

Hyvä perusperiaate on käyttää **using**-lausetta mahdollisimman lähellä sitä aluetta, jossa sen on tarkoitus olla voimassa (yleensä koodilohko tai funktio). Toisaalta paljon käytetyissä rakenteissa on usein dokumentaation kannalta selkeämpää kirjoittaa **using** heti siihen liittyvän otsikkotiedoston #include-käskyn jälkeen. Tämä suositus taas on heti ristiriidassa edellisen kappaleen "sotkemissäännön" kanssa. Nimiavaruudet ovat C++:ssa sen verran uusi asia, että parasta mahdollista suositusta selkeyden ja turvallisuuden kannalta tässä asiassa on vaikea antaa. Hyvä kompromissi on luottaa kääntäjän std-kirjastojen olevan oikein toimivia, jolloin niiden esittelyiden väliin voi kirjoittaa huoletta **using**-lauseita, mutta omissa osissa ja ostettujen kirjastojen kanssa kirjoittaa **using**-lauseet vasta kaikkien esittelyiden (otsikkotiedostojen) jälkeen. Listaus 1.11 seuraavalla sivulla on esimerkki tästä "yhdistelmäsäännöstä".

Kun normaali **using**-lause nostaa näkyville yhden nimen, sen erikoistapaus **using namespace** nostaa nimiavaruuden sisältä ***kaikki*** nimet nykyiselle tasolle ja siten käytännössä poistaa nimiavaruuden käytöstä kokonaan. Erityisesti lausetta **using namespace** std tulisi ai-

```
1  // Omat rakenteet esitellään std-kirjastoja ennen
2  // (tämä siksi että saamme tarkastettua niiden sisältävän kaikki
3  // tarvittavat #include-käskyt ts. ne ovat itsenäisesti kääntyviä yksikköjä)
4  #include "paivays.hh"
5  #include "swbus.hh"
6  #include "tietokanta.hh"
7  #include "loki.hh"
8  // Kaikista yleisimmin tässä tiedostossa käytetyt std-rakenteet esitellään
9  // heti niihin liittyvän otsikkotiedoston jälkeen
10 #include <iostream>
11 using std::cout;
12 using std::endl;
13 #include <vector>
14 using std::vector;
15 #include <string>
16 using std::string;
17 // lopuksi omiin moduuleihin liittyvät using-lauseet
18 using Paivays::PVM;
19 using Loki::varoitus;
20 using Loki::virhe;
```

LISTAUS 1.11: `using`-lauseen käyttö eri otsikkotiedostojen kanssa

na välttää C++-ohjelmissa. Tätä rakennetta käytetään kyllä usein opetuksessa ja lyhyissä esimerkkiohjelmissa selkeyden saavuttamiseksi, mutta suurissa C++-ohjelmistoissa emme sitä suosittele.

1.5.8 Nimeämätön nimiavaruus

Nimiavaruuksiin on määritelty erikoistapaus **nimeämätön nimiavaruus** (*unnamed namespace*), jolla rajoitetaan funktioiden ja muuttujien nimien näkyvyys yhteen käännösyksikköön (tämä tehtiin aikaisemmin C- ja C++-kielissä **`static`**-avainsanan avulla). Nimeämättömällä nimiavaruudella voidaan dokumentoida ne osat moduulin ohjelmakoodia, jotka ovat olemassa ainoastaan sen sisäistä toteutusta varten. Samalla kääntäjä myös pitää huolen, ettei niitä vahingossa pystytä käsittelemään moduulin ulkopuolelta.

Listauksessa 1.12 seuraavalla sivulla määritellään yksi muuttuja ja funktio käännösyksikön paikallisiksi nimeämättömällä nimiavaruudella. Tämän nimiavaruuden sisällä olevat rakenteet ovat samassa käännösyksikössä (tiedostossa) suoraan käytettävissä ilman näkyvyystarkenninta (jota nimeämättömällä nimiavaruudella ei tietysti

edes ole), mutta ne eivät ole käytettävissä käännösyksikön ulkopuolella.

Nimeämätön nimiavaruus suojaa sen sisällä olevat nimet tavallisen nimiavaruuden tapaan. Vaikka useassa käännösyksikössä on samoilla nimillä olevia rakenteita, niin ne eivät sotke toisiaan, kunhan kaikki ovat nimeämättömän nimiavaruuden sisällä. Tämän voi ajatella tapahtuvan siten, että kääntäjä tuottaa kulissien takana uniikin nimen jokaiselle nimeämättömälle nimiavaruudelle. Jos nimeämätöntä nimiavaruutta käyttää tavallisen nimiavaruuden sisällä, on nimeämättömän nimiavaruuden sisältö käytettävissä ainoastaan tämän tavallisen nimiavaruuden sisällä (esim. moduulin sisäinen funktio tai tietorakenne).

```
1  static unsigned long int laskuri; // Vanha tapa
2
3  // ISOC++:n mukainen tapa
4  // nimeämätön nimiavaruus:
5  namespace {
6    unsigned long int viiteLaskuri;
7    void lisaaViiteLaskuria() {
8      ++viiteLaskuri;
         ⋮
9    }
10 }
11
12 // Julkinen operaatio:
13 Pvm luoPaivaysOlio() {
14   lisaaViiteLaskuria();
15   // ylläoleva rutiinin kutsu toimii samassa tiedostossa ilman
16   // using-lausetta tai näkyvyystarkenninta.
17   // rutiinia ei pysty kutsumaan tiedoston ulkopuolisesta koodista.
       ⋮
18 }
19
```

LISTAUS 1.12: Nimeämätön nimiavaruus

1.6 Moduulituki muissa ohjelmointikielissä

Modulaarisuus on jo vanha ja hyväksi havaittu ohjelmointikielten ominaisuus, joten ei ole yllättävää, että sitä tuetaan tavalla tai toisella monissa nykyisissä ohjelmointikielissä. Seuraavassa esittelemme lyhyesti Modula-3- ja Java-kielten tukea ohjelman jakamisessa moduuleihin.

1.6.1 Modula-3 [Böszörményi ja Weich, 1996]

Modula-3-kielen yksi tärkeimmistä suunnitteluperusteista on ollut modulaarisen ohjelmoinnin tukeminen. Moduulien julkinen rajapinta kirjoitetaan erilliseen tiedostoon, jossa on näkyvissä ainoastaan moduulin tarjoamien palveluiden käyttämiseen tarvittavat tiedot. Tämä näkyy kuvasta 1.13 seuraavalla sivulla. **Moduulin käyttäjälle** on rajapinnan esittelyssä tarpeeksi informaatiota Päiväys-moduulin tarjoaman tietotyypin (`T`) käyttämiseen. **Moduulin tekijä** on ainoa, jonka kirjoittamassa toteutuksessa “paljastetaan” päiväystietorakenteen koko muoto.

Modula-3:n tapa esittää “vajaita” tyyppimäärittelyjä on hyvin pitkälle vietyä moduulin käyttäjän kannalta turhan tiedon kätkemistä. Ohjelmamoduuleja käsittelevän kääntäjäohjelman on tietysti tunnettava tietotyypin `Päiväys.T` rakenne ja sen viemä muistinvaraus, mutta tämä käännöstekninen yksityiskohta on piilotettu ohjelmoijalta.

1.6.2 Java [Arnold ja Gosling, 1996]

Javassa ohjelman nimet jaetaan kokonaisuuksiin tavalla, joka on hyvin lähellä C++:n nimiavaruuksia. Tästä on esimerkki listauksessa 1.14 seuraavalla sivulla. Java-kooditiedoston alussa voidaan avainsanalla `package` kertoa, minkä nimiseen pakkaukseen tiedostossa esiteltävät nimet kuuluvat. Javassa pakkausten nimet muodostavat hierarkian pisteellä erotelluilla osanimillä. Nimet otetaan käyttöön omassa ohjelmakoodissa `import`-lauseella. Oletuksena käyttöön tulee vain yksi nimi, mutta jos haluaan käyttää kaikkia pakkauksen julkisia nimiä, käytetään merkintää tähti (`.*`).

Javassa on myös rakenne `interface`, mutta se ei ole moduulien rajapintaesittely. Kyseisellä rakenteella esitellään lupaus määrätyn-

```
.................................. esittely ..................................
1 INTERFACE Päiväys;
2
3 TYPE T <: REFANY; (* Kerrotaan vain, että T on viite dataan, jonka
4                      rakennetta moduulin käyttäjän ei tarvitse tietää *)
5
6 PROCEDURE Luo( paivä, kuukausi, vuosi : INTEGER ) : T;
7 PROCEDURE Tulosta( kohde : T );
8 ...
9 END Päiväys.
.................................. käyttö ..................................
1 MODULE Käyttö;
2
3 IMPORT Päiväys;
4
5 VAR vappu : Päiväys.T;
6 BEGIN
7   vappu := Päiväys.Luo( 1, 5, 2001 );
8   Päiväys.Tulosta( vappu );
.................................. määrittely ..................................
1 MODULE Päiväys;
2
3 REVEAL T = BRANDED REF RECORD
4               päivä, kuukausi, vuosi : INTEGER;
5           END;
6 ...
```

LISTAUS 1.13: Moduulituki Modula-3-kielessä

```
1  package com.jyke.oliokirja.heippa;
2
3  import java.applet.Applet; // käytetään luokkaa Applet
4  import java.awt.*; // awt-moduulista käyttöön kaikki
5
6  public class Heippa extends Applet {
7    public void paint(Graphics g) // java.awt.Graphics
8    {
9      g.drawString("Heippa!", 0, 0);
10   }
11 }
```

LISTAUS 1.14: Moduulituki Java-kielessä

laisesta rajapinnasta, jonka jokin luokka voi toteuttaa (katso aliluku 6.9). Moduulit ja niiden rajapinnat ovat tätä laajempi käsite.

Luku 2

Luokat ja oliot

Lucius: "Ay me! This object kills me."
– The Tragedy of Titus Andronicus, Act 3, Scene 1 [Shakespeare, 1593]

Ohjelmointikielten oliokäsite on peräisin Simula-67 -ohjelmointikielestä, joka kehitettiin erilaisten simulointimallien toteutusta varten [Sethi, 1996]. Kun ongelmana on esimerkiksi simuloida pankissa asioivien asiakkaiden keskimääräistä odotusaikaa, Simula-ohjelmassa kuvataan simuloitavan ongelman kannalta oleellisia rakenteita ohjelmointikielen rakenteina, joita nimitettiin **olioiksi**: asiakas, jono asiakkaita, toimihenkilö. Simulointiohjelma "pyöritteli" näitä olioita muistissaan: asiakasolio tulee pankkiin, menee jonoon, odottaa vuoroaan, menee palveltavaksi, kertoo asiansa A, jonka palveluun menee aikaa T, kiittää ja poistuu simulaatiosta. Vaikka todelliset asiakkaat ovat yksilöitä, ohjelman mallintamat abstraktit asiakasoliot sisältävät paljon yhteisiä piirteitä. Kuvaamalla nämä yhteiset piirteet ohjelmassa vain kerran saadaan säästetyksi aikaa ja vaivaa. Samat piirteet omaavat oliot kuuluvat samaan **luokkaan** ja niillä kaikilla on luokan määrittelemät ominaisuudet ja toiminnallisuus.

2.1 Olioiden ominaisuuksia

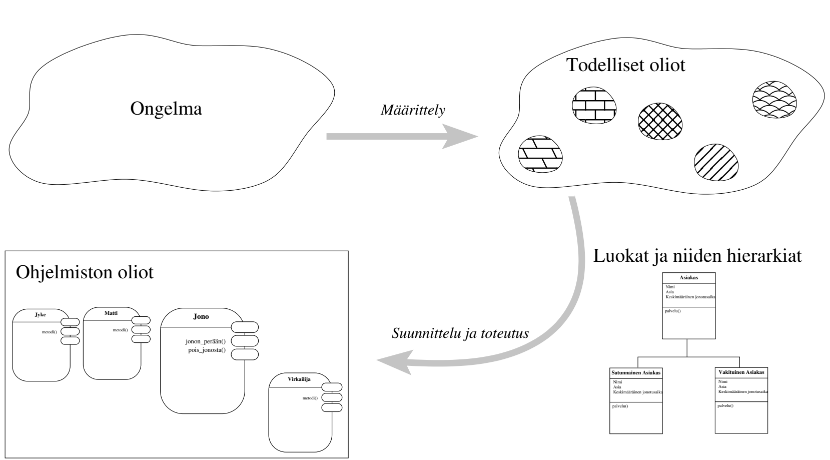
Olioita voidaan tarkastella usealta näkökannalta. Kuten moduuleja voidaan olioitakin tarkastella ulkoapäin (olion käyttäjä) tai sisältä kä-

sin (olion suunnittelija ja toteuttaja). Olion julkinen rajapinta kertoo käyttäjälle, mitä tehtäviä se lupaa toteuttaa vastuualueellaan.

Olioiden käyttötarkoitus on kaksijakoinen: niillä pyritään mallintamaan ongelmaa (määrittelyssä), ja toisaalta ne pyrkivät tarjoamaan käytännöllisen tavan ohjelmiston toteuttamiseen (suunnittelu ja toteutus). Näiden "todellisten" ja ohjelmallisten olioiden ero on näkyvissä kuvassa 2.1. Olio-ohjelmien tekeminen ei kuitenkaan ole missään nimessä tämän kuvan nuolten mukainen suoraviivainen prosessi, jossa toteutetaan vain todelliset oliot jollain olio-ohjelmointikielellä. Aina kaikki määrittelyssä löydetyt "todelliset" oliot eivät ole olioita toteutetussa ohjelmassa, ja ohjelmiston toteutuksessa usein tarvitaan olioita, joita määrittelyssä ei ole olemassa (toteutusta tukevat oliot).

2.1.1 Oliolla on tila

Jokaisella oliolla on aina olemassa **tila**, jonka sisältämä informaatio on oleellinen ohjelmiston toiminnan kannalta. Kirjastonkirja-olio voi esimerkiksi sisältää tiedon kirjan nimestä, ISBN-numerosta ja han-



KUVA 2.1: Ongelmasta ohjelmaksi olioilla

kintapäivämäärän. Suunnittelussa olion tila koostuu attribuuteista, joilla kerrotaan mitä informaatiota olion vastuualueen toteutuksessa tarvitaan. Toteutuksessa nämä attribuutit toteutetaan ohjelmointikielen tarjoamilla tavoilla, joita voivat olla muuttujat, tietueet tai oliot — olion sisäisessä toteutuksessa on hyvin tavallista käyttää toisia (mahdollisesti muualta hankittuja kirjastoituja) olioita hyväksi. Kirjastonkirja-olio voi toteutuksessa siis sisältää merkkijonon (kirjan nimi), numerotietueen (ISBN) ja päivämäärä-olion.

Olion tila muuttuu ohjelman suorituksen aikana ja usein tämä tila kapseloidaan piiloon olion sisälle siten, että sitä voidaan tarkastella ja muuttaa ainoastaan olion julkisen rajapinnan tarjoamien palveluiden kautta. Yleensä pidetään suunnittelu- tai ohjelmointivirheenä tilannetta, jossa ohjelmassa olevilla olioilla ei ole määriteltyä tilaa jonain suoritusajankohtana.

2.1.2 Olio kuuluu luokkaan

Kirjastonkirja-olioiden tilaan kuuluu nimi (merkkijono, max 256 merkkiä), ISBN (määrämuotoinen tietue numeroita) ja hankintapäivä (päiväys-olio). Jokaisella kirjalla on tietysti omat tietonsa, mutta attribuuttien tietotyypit ovat kaikilla Kirjastonkirjoilla samat ja nämä voidaan määritellä keskitetysti yhdessä paikassa (luokka). Vastaavasti palvelut (julkinen rajapinta) on jokaisella Kirjastonkirjalla sama ja nekin kannattaa määritellä vain kerran. Luokka voidaan ajatella rakenteeksi, joka edustaa kaikkia samalla tavoin rakentuneita (attribuutit, palvelut ja käyttäytyminen) olioita. Luokka myös määrää miten kyseisen tyyppisiä olioita luodaan ja tuhotaan ohjelmassa.

Jos kerran olio on mallinnuksen ja toteutuksen perusyksikkö, niin miksi puhua mistään luokista? Täsmälleen samasta syystä kuin aikaisemmin modulaarisuuden puolustuspuheessa perusideana on asioiden yksinkertaistaminen — abstrahointi. Ohjelmointikielen luokka kertoo “yleisen rakenteen” yksittäisen olion tilalle ja määrittelee rajapinnat olion käyttämiselle. Useampaa oliota edustava luokka on olio-ohjelmien suunnittelussa ehkä eniten käytetty rakenne. Ohjelmistojen rakennetta kuvaavat piirrokset ovat usein luokkakaavioita ja puhuvat luokkien ominaisuuksista, eivät yksittäisistä olioista.

Kun ohjelmassa luodaan uusi olio, puhutaan olion **instantioinnista**, jolloin olion luokka määrää muodostuvan uuden olion rakenteen (olion kuluttaman muistin määrän sekä olion tilan sisäisen raken-

teen) ja uuden olion alkutilan **alustustoimien** avulla (joita käsitellään luvussa 3).

Olioita voidaan verrata ohjelmointikielen muuttujiin, joille yhteiset asiat kertoo muuttujan tietotyyppi. Olioilla tyyppiä vastaa olion luokka. Luokka on olio-ohjelmien perusyksikkö, jonka varaan laajemmat ja vahvemmat ominaisuudet rakentuvat.

Olio-ohjelmien uudelleenkäytettävyyttä lisää ominaisuus, jossa olemassa olevaan luokkaan voidaan lisätä tai muuttaa ominaisuuksia (**periytyminen**, luku 6). Luokat voivat muodostaa keskinäisiä hierarkioita, joissa samaan "kokoelmaan" kuuluvien luokkien oliot voivat toimia yhtenäisesti ja tarvittaessa hieman toisistaan poikkeavalla tavalla (periytyminen ja **polymorfismi**, aliluku 6.1). Luokista itsestään voidaan tehdä yleiskäyttöisiä malleja ("metaluokkia"), joista muodostetaan malliin sopivia luokkainstansseja (**geneerisyys**, luku 9). Useimmat olio-ohjelmointikielet tuottavat ohjelmia, joissa jokainen olio tietää mihin luokkaan se kuuluu. Lähes kaikki olio-ohjelmointikielet tarjoavat myös mekanismin, jolla ajoaikana voidaan tarkastella, kuuluuko olio määrättyyn luokkaan (C++:n RTTI on käsitelty aliluvussa 6.5.3).

Luokka on käsite, joka esiintyy ainoastaan ohjelman suunnittelu- ja toteutusvaiheessa. Ohjelmiston suorituksen aikana on olemassa vain olioita.[ϒ] Tämä näkökulma hämärtyy helposti, koska varsin usein olio-ohjelman suunnitteluvaiheessa on näkyvissä vain ohjelman luokkarakenne. Luokka on kuvaustapa, jolla voidaan puhua samaan "ryhmään" eli luokkaan kuuluvien olioiden yhteisistä ominaisuuksista, ja lopullisessa ohjelmistossa on aina toimimassa luokan mallin toteuttavia olioita.

2.1.3 Oliolla on identiteetti

Koska olion sisäinen tila on muista olioista riippumaton, on täysin mahdollista, että jonain ajanhetkenä kahdella tai useammalla saman luokan oliolla on täsmälleen sama tila. Ajatellaan luokka, joka kuvaa päivämääriä. Voimme luoda tämän mallin mukaisesti oliot A ja B, jotka molemmat kuvaavat samaa päivämäärää (vaikkapa 22.4.2069). Kuinka voimme erottaa nämä oliot toisistaan, kun molempien päivämääräattribuutilla on sama arvo? Kysymys voi kuulostaa oudolta,

[ϒ] Poikkeuksena ovat erityistapaukset, joissa halutaan kerätä oliojoukon ominaisuuksia yhteen "luokkaolioksi" (aliluku 8.2.1)

koska teksti itsessään sisältää tiedon erottelusta ("oliot A ja B"). Oleellista on huomata, että tässä on jokin olioiden ***ulkopuolinen*** tapa (nimeäminen), jolla ne erotetaan toisistaan.

Ohjelmiston toteuttamiseksi on välttämätöntä, että jokainen olemassa oleva olio voidaan yksikäsitteisesti tunnistaa ja käsitellä. Tätä tarkoitusta varten olio-ohjelmointikielet tarjoavat **identiteetin** (*identity*) käsitteen.

Yksinkertaisimmillaan identiteetti voi olla muuttujanimi, joka kertoo mistä oliosta puhutaan ("muuttujassa A tai B"). Identiteetti voi olla myös esim. muistiosoite, joka kertoo missä olio sijaitsee muistissa. Oikein toimiva järjestelmä ei talleta olioita muistiin päällekkäin, joten muistiosoite erottelee kaikki oliot toisistaan eli antaa niille yksilöllisen identiteetin.

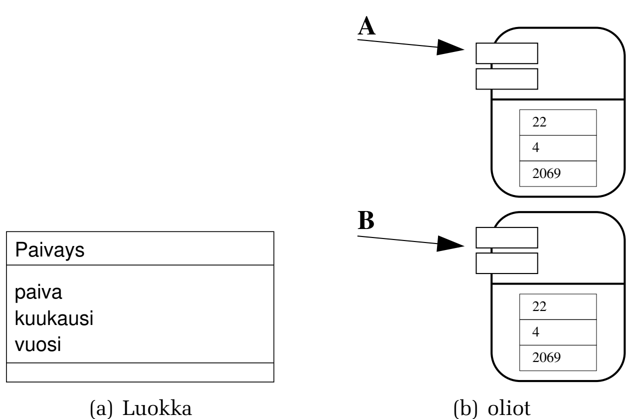
Mutkikkaammissa nykyaikaisissa järjestelmissä voi olioita olla taltioituna tietokannoissa tai niitä voidaan käsitellä hajautetusti tietoverkossa, jolloin yksikäsitteinen identiteetti (järjestelmän- tai jopa maailmanlaajuisena) on välttämätön olioiden käytössä.

Identitettiä voidaan käyttää tarkistamaan tarvittaessa, onko kyseessä sama olio ilman että olion tila vaikuttaa tulokseen (tai että sitä edes tarvitsee tarkistaa). Esimerkissä voidaan sanoa olioiden A ja B olevan yhtä suuria (edustavan samaa päivämäärää) mutta ne eivät ole sama olio, koska identiteetit ovat erilaiset ("A ei ole B"). Olioiden eri ominaisuuksia (tila, identiteetti ja luokka) on havainnollistettu kuvassa 2.2 seuraavalla sivulla.

2.2 Luokan dualismi

Olio-ohjelmointikielen luokalla voidaan katsoa olevan kaksi esiintymismuotoa tai katselukulmaa: moduuli ja tyyppi. Luokka sisältää niiden molempien ominaisuuksia ja pystyy esiintymään ohjelmassa kumman tahansa normaalissa käyttötarkoituksessa.

- **Luokka moduulina.** Luokka toteuttaa rajapintojen ja tiedon kätkennän periaatteet: luokan suunnittelija määrittelee julkiseen rajapintaan ne operaatiot, joilla luokasta tehtyä oliota voidaan käsitellä. Kaikki pelkästään luokan toteutukseen liittyvät ominaisuudet (tiladata ja sisäiset funktiot) pystytään kätkemään luokan käyttäjältä.



(a) Luokka (b) oliot

KUVA 2.2: Päiväys-luokka ja siitä tehdyt oliot A ja B

- **Luokka tietotyyppinä.** Luokka esiintyy olio-kielessä “ykkösluokan kansalaisena” kielen perustietotyyppien (esim. liukuluvut) kanssa. Kaikki muuttujille tutut operaatiot (esim. laskutoimitukset, sijoitukset ja arvojen välitys parametreina) voidaan määritellä myös luokasta tehtyjen olioiden yhteydessä.

Tämä kahtiajako nähdään myös silloin, jos ajattelemme jo ennen oliokäsitettä olleita tietotyyppejä “olioina”. Esimerkiksi useimmissa ohjelmointikielissä käytettävissä oleva liukulukutyyppi sisältää nämä kaksi näkökulmaa:

- **Liukuluku moduulina.** Automaattisesti oletamme, että liukuluvuilla on olemassa laskuoperaatioita (ainakin yhteen-, vähennys-, kerto- ja jakolasku), lukuja voidaan alustaa määrättyyn arvoon ja vertaamalla liukulukuja saadaan niiden välille suuruusjärjestys. Oliomaailmassa nämä operaatiot ovat liukulukuluokkaan kuuluville olioille määritellyn julkisen rajapinnan palveluita. Liukulukua käyttävän ohjelmoijan ei tarvitse tuntea käytetyn laiteympäristön liukuluvun esitysmuotoa konekoodin tasolla (esim. IEEE-754). Tämä toteutusyksityiskohta on kätketty liukulukumoduulin sisälle.

- **Liukuluku tietotyyppinä.** Voimme esimerkiksi tehdä liukutyyppiä olevia muuttujia, sijoittaa niitä toisiin muuttujiin ja välittää muuttujien arvoja parametreina. Tämä olioiden käyttö muuttujina on tärkein ero staattista rakennetta edustavien moduulien ja luokista tehtyjen olioiden välillä.

2.3 C++: Luokat ja oliot

Jo aiemmin on mainittu, että yksi tapa ajatella olioita ja luokkia on jakaa niiden sisältö kahteen osaan — toimintoihin ja tietorakenteisiin — ja ajatella olioita ikään kuin tietorakenteina, joihin on "liitetty" niihin kohdistuvat toiminnot.

Tämä ajattelutapa on ilmeisesti ollut lähtökohtana, kun C++-kieltä on kehitetty C-kielestä [Stroustrup, 1994]. C++:n luokat muistuttavat joiltain osin erittäin paljon C:n **`struct`**-tietotyyppejä (jopa niin paljon, että C++:ssa *itse kielen kannalta* avainsanat **`struct`** ja **`class`** ovat lähestulkoon vaihtokelpoisia). Aivan kuten **`struct`**-tietotyypitkin, myös luokat täytyy C++:ssa *esitellä* jokaisessa käännösyksikössä ennen kuin niitä voi käyttää. Tietotyypeistä poiketen luokilla on kuitenkin myös sisäinen toteutuksensa — rajapinnan palvelut toteuttava ohjelmakoodi tai toisin sanottuna rajapintafunktioiden toteutus —, joka kirjoitetaan tyypillisesti erikseen omaan tiedostoonsa (tai tiedostoihinsa).

2.3.1 Luokan esittely

Luokan käyttäjälle riittää yleensä **luokan esittely**. Luokan esittely kertoo luokasta kaiken luokan käyttöön tarvittavan. Ohjelmoijalle se kertoo luokan käyttöön tarvittavan rajapinnan. Kääntäjää varten luokan esittely sisältää tietoa luokan periytymissuhteista sekä tietoa luokan sisällöstä muistinvarausta yms. varten. Kuten esittelyt yleensä, myös luokan esittelyt kirjoitetaan tyypillisesti otsikkotiedostoon, josta luokan käyttäjä sitten lukee ne **`#include`**-komennolla omaan kooditiedostoonsa.

Luokan esittely alkaa avainsanalla **`class`**, jonka jälkeen tulee esiteltävän luokan nimi. Tämän jälkeen kerrotaan aaltosuluissa, mitä luokka sisältää. Luokan sisältö voi koostua seuraavista asioista:

- **Jäsenmuuttujat** ovat luokan olion sisällä olevia muuttujia. Jäsenmuuttujia käytetään olion sisäisen tilan muistamiseen. Jäsenmuuttujista kerrotaan aliluvussa 2.3.2.

- **Jäsenfunktiot** ovat funktioita, joihin kirjoitetaan luokan toiminnallisuus. Luokan rajapinta koostuu jäsenfunktioista. Rajapinnan lisäksi luokalla voi olla myös jäsenfunktioita, jotka eivät näy ulospäin. Jäsenfunktioita käsitellään aliluvussa 2.3.3.

- **Sisäiset tyypit** ovat luokan rajapintaan tai sisäiseen toteutukseen liittyviä tyyppimäärittelyjä. Niistä kerrotaan tarkemmin aliluvussa 8.3.

- **Luokkamuuttujat** ovat luokan kaikille olioille yhteisiä muuttujia. Niitä voidaan käyttää sellaisen tiedon tallettamiseen, joka ei ole oliokohtaista, vaan koskee kaikkia luokan olioita yhdessä. Luokkamuuttujat käsitellään tarkemmin aliluvussa 8.2.2.

- **Luokkafunktiot** ovat luokkamuuttujien vastine funktioiden puolella. Ne ovat funktioita, joiden toiminnallisuus ei koske yksittäistä oliota, vaan koko luokkaa kokonaisuutena. Yleensä luokkafunktioita käytetään luokkamuuttujien käsittelyyn. Luokkafunktioista kerrotaan aliluvussa 8.2.3.

Listaus 2.1 seuraavalla sivulla sisältää esimerkkinä yksinkertaisen luokan `PieniPaivays` esittelyn, joka on kirjoitettu tiedostoon `pienipaivays.hh`. Huomaa, että tämä luokkaesimerkki on tarkoituksella yksinkertaistettu, ja siitä puuttuu useita C++:n luokille tarpeellisia asioita (kertaustehtävä: etsi luokkaesimerkin puutteet ☺).

On syytä huomata, että tässä teoksessa käytetty termi “luokan esittely” ei kirjaimellisesti vastaa englanninkielistä C++-terminologiaa, jossa samasta asiasta käytetään nimitystä “*class definition*” (kirjaimellisesti “luokan *määrittely*”). Sana “esittely” kuvaa tilannetta kuitenkin ehkä paremmin, koska esittelyn yhteydessä ei kerrota luokan rajapintafunktioiden toteutusta.

2.3.2 Jäsenmuuttujat

Jäsenmuuttujista (*data member*) käytetään myös nimityksiä “attribuutti”, “kenttä”, “instanssimuuttuja” ja “tietojäsen”. Ne ovat olioon kiinteästi liittyviä muuttujia, jotka muodostavat olion sisäisen tilan.

```
1  #ifndef PIENIPAIVAYS_HH
2  #define PIENIPAIVAYS_HH
3
4  class PieniPaivays
5  {
6  public:
7    void asetaPaivays(unsigned int p, unsigned int k, unsigned int v);
8    void sijoitaPaivays(PieniPaivays& p);
9    void asetaPaiva(unsigned int paiva);
10   void asetaKk(unsigned int kuukausi);
11   void asetaVuosi(unsigned int vuosi);
12
13   unsigned int annaPaiva();
14   unsigned int annaKk();
15   unsigned int annaVuosi();
16
17   void etene(int n);
18   int paljonkoEdella(PieniPaivays& p);
19
20 private:
21   unsigned int paiva_;
22   unsigned int kuukausi_;
23   unsigned int vuosi_;
24 };
25
26 #endif
```

LISTAUS 2.1: Esimerkki luokan esittelystä, `pienipaivays.hh`

Tyypiltään jäsenmuuttujat voivat olla mitä tahansa — kokonaislukuja, osoittimia, viitteitä tai vaikkapa toisia olioita.

Jäsenmuuttujat täytyy C++:ssa esitellä luokan esittelyn sisällä, ja niiden esittely muistuttaa suuresti tavallisten muuttujien esittelyä. Listauksen 2.1 riveillä 21–23 on esitelty päiväysluokan tarvitsemat jäsenmuuttujat, joihin päiväysolion sisältämä päiväystieto talletetaan. Jäsenmuuttujien nimeämisessä ei C++:n kannalta ole mitään erityisiä sääntöjä, mutta useissa ohjelmointityyleissä (katso liite B) jäsenmuuttujat nimetään niin, että ne on koodissa helppo erottaa esimerkiksi jäsenfunktioiden parametreista.

Tässä teoksessa käytetään C++:ssa kohtalaisen yleistä nimeämistapaa, jossa jäsenmuuttujien nimien perään lisätään alaviiva. Vastaava englanninkielisessä ohjelmoinnissa yleinen käytäntö on lisätä jäsenmuuttujien eteen sana “my”, siis `myDay`, `myMonth`, `myYear` ja niin edel-

leen. Sen sijaan joissain teoksissa näkyvä tapa lisätä jäsenmuuttujien nimien *eteen* alaviiva (_paiva) ***ei*** ole suotava, koska C++-standardi varaa tietyt alaviivoilla alkavat nimet kääntäjän sisäiseen käyttöön.

Olio-ohjelmoinnissa on varsin tavallista, että olion jäsenmuuttujat ovat tyypiltään toisia olioita. Päiväystyyppistä oliota voisi esimerkiksi käyttää jäsenmuuttujana luokassa, joka kuvaa kirjaston tietojärjestelmässä yhden kirjan tietoja, joihin kuuluu myös palautuspäivä. Osa tällaisen luokan esittelystä voisi olla seuraavanlainen:

```
class Kirja
{
   ⋮
private:
  string nimi_; // C++:n merkkijonoluokka, ks. liitteen A aliluku A.4
  string tekijannimi_;
  PieniPaivays palautuspvm_;
   ⋮
};
```

Tällainen hierarkkinen rakenne, jossa olio sisältää jäsenmuuttujanaan olion, jolla taas on sisällään omat jäsenmuuttujansa, on hyvin tyypillinen olio-ohjelmoinnissa. Sen avulla olion sisältämä data voidaan abstrahoida ja kapseloida selkeiksi kokonaisuuksiksi, ja olioiden rajapintojen avulla myös tiedon käsittely voidaan jakaa selkeisiin osiin luokkien avulla.

Koska jäsenmuuttujat liittyvät kiinteästi olioon, niiden elinkaari on täsmälleen sama kuin olionkin. Kun olio syntyy, syntyvät myös kaikki sen jäsenmuuttujat. Samoin jäsenmuuttujat tuhoutuvat aina samalla kuin itse oliokin. Kaikki olion sisäinen tieto ei kuitenkaan ole luonteeltaan sellaista, että se olisi saatavilla heti olion syntymän yhteydessä tai että se kestäisi olion tuhoutumiseen saakka. Esimerkiksi listaoliolla olisi järkevää olla jäsenmuuttujissaan tallessa listan sisältämät alkiot. Tämä ei kuitenkaan ole suoraan mahdollista, koska listan alkiot eivät ole tiedossa listaolion syntyessä ja vastaavasti niitä voidaan poistaa ja lisätä listaolioon koska tahansa. Tällaisissa tapauksissa on C++:ssa tapana laittaa olioon jäsenmuuttujaksi osoitin tai osoittimia, joiden päähän voi sitten myöhemmin sijoittaa tietoa, joka ei ole olion “omistuksessa” koko olion elinaikaa. Yleensä tällaisissa tapauksissa tarvitaan myös olioiden tai datan dynaamista luo-

mista, josta kerrotaan enemmän aliluvussa 3.3.4. Olioiden elinkaarta käsitellään tarkemmin luvuissa 3 ja 7.

2.3.3 Jäsenfunktiot

Jäsenfunktioita (*member function*) kutsutaan myös monilla muilla nimillä olio-ohjelmoinnissa. Yleisimpiä ovat "metodi", "rutiini", "palvelu" tai "operaatio". Kaikki nämä nimitykset kuvaavat saman asian eri puolia. Jäsenfunktiot ovat funktioita, jotka toteuttavat olion tarjoamat palvelut ja joiden avulla eri oliot kommunikoivat keskenään. Kaikki luokan toiminnallisuus kirjoitetaan jäsenfunktioihin (ja luokkafunktioihin, joista enemmän aliluvussa 8.2.3).

Kuten tavalliset funktiotkin, myös jäsenfunktiot täytyy esitellä ennen kuin niitä voi käyttää. Rivit 7–18 listauksessa 2.1 sivulla 62 sisältävät luokan `PieniPaivays` jäsenfunktioiden esittelyt luokan esittelyn sisässä. Jäsenfunktioiden esittelyssä syntaksi on lähes täsmälleen sama kuin tavallistenkin funktioiden esittelyssä.

Jäsenfunktion esittelyssä kerrotaan vain, miltä jäsenfunktio näyttää ulospäin eli miten sitä voi kutsua. Tavallisten funktioiden tapaan jäsenfunktioiden varsinainen toteutus (koodi) kirjoitetaan erilliseen kooditiedostoon. Yleinen käytäntö on, että jokaista luokkaa kohti kirjoitetaan yksi jäsenfunktioiden toteutukset sisältävä kooditiedosto, mutta C++ ei mitenkään rajoita tätä. Joskus saattaa olla tarpeen hajottaa luokan toteutus useisiin tiedostoihin, joskus taas on järkevää kirjoittaa muutaman toisiinsa tiiviisti liittyvän luokan toteutus samaan kooditiedostoon — tällöin usein myös luokkien esittelyt kirjoitetaan samaan otsikkotiedostoon.

Listaus 2.2 seuraavalla sivulla sisältää osan `PieniPaivays`-luokan jäsenfunktioiden toteutuksista, jotka ovat tiedostossa `pienipaivays.cc`. Tiedoston alussa luetaan ensin sisään luokkaesittely tiedostosta `pienipaivays.hh`. Tämä on aina tehtävä kooditiedostoissa ennen jäsenfunktioiden määrittelyjä, jotta kääntäjä saa luettua esittelystä tarvitsemansa tiedot luokasta.

Jäsenfunktioiden määrittely on syntaksiltaan hyvin samanlainen kuin tavallisen C++:n funktion määrittely. Näkyvin ero on, että määrittelyn alussa käytetään näkyvyystarkenninta `::`, eli jäsenfunktion nimi esiintyy aina muodossa `Luokannimi::jfunktionnimi`. Tämä on tarpeen, jotta kääntäjä tietää, minkä luokan jäsenfunktiota ollaan mää-

```
 1 #include "pienipaivays.hh"
 2
 3 void PieniPaivays::asetaPaivays(unsigned int p, unsigned int k,
 4                                   unsigned int v)
 5 {
 6   asetaPaiva(p);
 7   asetaKk(k);
 8   asetaVuosi(v);
 9 }
10
11 void PieniPaivays::asetaPaiva(unsigned int paiva)
12 {
13   paiva_ = paiva;
14 }
15
16 unsigned int PieniPaivays::annaPaiva()
17 {
18   return paiva_;
19 }
      ⋮
```

LISTAUS 2.2: Jäsenfunktioiden toteutus, `pienipaivays.cc`

rittelemässä — useissa luokissa kun voi olla samannimisiä jäsenfunktioita.

Toinen ero tavallisiin funktioihin on, että jäsenfunktion koodissa voi viitata "oman olion" jäsenmuuttujiin ja toisiin jäsenfunktioihin suoraan. Tämä on mahdollista, koska jäsenfunktioita kutsutaan aina jonkin olion kautta, joten jäsenfunktio "tietää" kutsun yhteydessä, minkä olion palvelua ollaan suorittamassa. Niinpä listauksen 2.2 rivillä 6 kutsutaan *oman olion* `asetaPaiva`-jäsenfunktiota, joka puolestaan sijoittaa parametrina tulleen päiväyksen talteen *oman olion* jäsenmuuttujaan `paivays_` rivillä 13. Vastaavasti rivillä 18 jäsenfunktio `annaPaiva` palauttaa paluuarvonaan *oman olion* jäsenmuuttujan `paiva_` arvon. Suora pääsy olion jäsenmuuttujiin ja jäsenfunktioihin on luontevaa, onhan jäsenfunktion toteutuksen tarkoitus nimenomaan tarjota olion käyttäjälle jokin palvelu operoimalla olion sisältämällä tiedolla ja mahdollisesti käyttämällä hyväkseen muita olion tarjoamia palveluita.

Joskus jäsenfunktioiden koodissa tulee tarve erikseen viitata olioon itseensä. Tyypillinen esimerkki tästä on tilanne, jossa jäsen-

funktio joutuu antamaan olion itsensä parametrina jollekin toiselle funktiolle. Tällaisia tilanteita varten jäsenfunktioiden koodissa voi käytää erityismerkintää **this**. Luokan X jäsenfunktion koodissa **this**in tyyppi on "osoitin X-olioon", eli se käyttäytyy aivan kuin se olisi esitelty lauseella "X* **this**;". Osoittimen arvo on automaattisesti sellainen, että se osoittaa olioon, jonka jäsenfunktion koodia ollaan suorittamassa, siis "olioon itseensä".

Jos esimerkiksi ohjelmassa on funktio rekisteroiPvm, joka ottaa parametrikseen osoittimen päiväysolioon, voi päiväysolion jokin jäsenfunktio rekisteröidä olion itsensä tietokantaan lauseella rekisteroiPvm(**this**);. Jos funktio ottaisikin osoittimen sijaan parametrikseen viitteen[☋] päiväysolioon, olisi syntaksi vastaavasti rekisteroiPvm(***this**);.

Joissain oliokielissä jäsenfunktioiden koodissa ei voi lainkaan käyttää jäsenmuuttujia tai jäsenfunktioita suoraan, vaan niihin täytyy aina viitata erikseen olion itsensä kautta, esimerkiksi syntaksilla **this**->jmuuttuja tai self.muuttuja. Vaikka C++:ssakin tämä olisi mahdollista **this**-merkinnän avulla, ei se ole tapana.

2.3.4 Luokkien ja olioiden käyttö C++:ssa

Kun kooditiedostossa luokan esittely on luettu sisään, voidaan luokasta luoda koodissa olioita samalla tavalla kuin normaaleja muuttujia luodaan C++:ssa. Muutenkin luokka käyttäytyy kuten mikä tahansa muukin käyttäjän itsensä määrittelemä tyyppi, kuten esim. **struct**-tietorakenne. Tässä suhteessa C++ heijastaa suoraan aliluvussa 2.2 mainittua "luokka tietotyyppinä" -periaatetta.

Listauksessa 2.3 sivulla 68 on esimerkki yksinkertaisesta pääohjelmasta, jossa käytetään aiemmin määriteltyä PieniPaivays-luokkaa. Jälleen tiedoston alussa luetaan sisään luokan esittely otsikkotiedostosta. Rivillä 14 luodaan luokasta PieniPaivays päiväysolio aivan kuten seuraavalla rivillä luodaan tyypistä **int** kokonaislukumuuttuja. Tämän jälkeen jäsenfunktioita kutsutaan syntaksilla olionNimi.jfunktio(parametrit), kuten tapahtuu rivillä 17. Jos olion sijaan käytetään osoitinta olioon, jäsenfunktioita kutsutaan syntaksilla osoitin->jfunktio(parametrit). Tämä näkyy rivillä 30. Jos taas käytetään viitettä olioon, näyttää jäsenfunktion kutsu samalta kuin

[☋] C++:n viitetyypin käyttö esitellään lyhyesti liitteen A aliluvussa A.1.

suoraan oliota käytettäessä. Tästä on esimerkkinä viitteen `pvm` kautta tapahtuva kutsu rivillä 40.

Olioita voi myös välittää parametreina toisiin funktioihin (ja jäsenfunktioihin). Listauksen rivi 27 näyttää funktion, joka ottaa parametrinaan osoittimen olioon. Vastaavasti rivillä 37 on funktio, joka ottaa parametrina viitteen olioon. Olioiden välittäminen "normaaleina" arvoparametreina on myös mahdollista, mutta se vaatii kopiorakentajan käsitteen tuntemista, jota käsitellään myöhemmin aliluvussa 7.1.2. Huomaa, että kun funktioon välittää osoittimen tai viitteen olioon, niin funktiossa osoittimen tai viitteen läpi tehtävät jäsenfunktiokutsut kohdistuvat alkuperäiseen olioon, aivan kuten normaalistikin osoitin- ja viiteparametreja käytettäessä.

```
1  #include "pienipaivays.hh"
2
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  // Funktio, joka tarkistaa, onko vuoden sisällä karkauspäivää
8  bool onkoKarkauspaivaa(PieniPaivays* pvm_p);
9  // Funktio, joka kertoo montako päivää on saman vuoden jouluun
10 int kauankoJouluun(PieniPaivays& pvm);
11
12 int main()
13 {
14   PieniPaivays kesapaiva;
15   int paiviaJouluun = 0;
16
17   kesapaiva.asetaPaivays(21,7,1999);
18   if (onkoKarkauspaivaa(&kesapaiva))
19   {
20     cout << "Karkauspäivä on alle vuoden päässä." << endl;
21   }
22   paiviaJouluun = kauankoJouluun(kesapaiva);
23   cout << "Jouluun on " << paiviaJouluun << " päivää." << endl;
24 }
25
26 // Karkauspäiväfunktion (typerä) toteutus

27 bool onkoKarkauspaivaa(PieniPaivays* pvm_p)
28 {
29   unsigned int vanhaPaiva = pvm_p->annaPaiva(); // Päivä talteen
30   pvm_p->etene(365); // Siirry 365 päivää eteenpäin
31   // Karkauspäivä on tullut vastaan, jos 365 päivää ei ollut koko vuosi
32   bool oliKarkauspaiva = (pvm_p->annaPaiva() != vanhaPaiva);
33   pvm_p->etene(-365); // Palauta vanha päivä menemällä takaisin
34   return oliKarkauspaiva;
35 }
36 // Jouluunlaskun toteutus

37 int kauankoJouluun(PieniPaivays& pvm)
38 {
39   PieniPaivays joulu;
40   joulu.asetaPaivays(24, 12, pvm.annaVuosi()); // Saman vuoden joulu
41   return joulu.paljonkoEdella(pvm); // Paljonko joulu on edellä?
42 }
```

LISTAUS 2.3: Esimerkki luokan käytöstä, `ppkaytto.cc`

Luku 3

Olioiden elinkaari

*Bernie: "But I did **okay**, didn't I? — I mean I got, what, fifteen thousand years. That's pretty good. **Isn't** it? I lived a pretty long time."*

Death ☥: "You lived what anybody gets, Bernie. — You got a lifetime. — No more. — No less. — You got a lifetime."

– *Brief Lives [Gaiman* ja muut*, 1994]*

"Normaalissa" ei-olio-ohjelmoinnissa muuttujien "elinkaaresta" ei yleensä ole tarpeen erikseen puhua — muuttujia luodaan kun niitä tarvitaan, ja käytetään niin kauan kuin käytetään. Ainoat muistettavat asiat ovat muuttujien alustus tarvittaessa sekä dynaamisesti varatun muistin vapauttaminen. Paikallisten muuttujien käyttö on vielä helpompaa, koska niitä ei tarvitse koskaan muistaa vapauttaa.

Olio-ohjelmoinnissa olioiden käyttäminen sen sijaan vaatii enemmän harkintaa. Oliot voivat olla monimutkaisiakin kokonaisuuksia ja niiden "syntymä ja kuolema" saattavat vaatia kaikenlaisia toimenpiteitä. Olio-ohjelmoinnin vastuualueajattelun mukaista olisi se, että olio itse vastaisi tällaisista toimenpiteistä, jotta ne eivät jäisi käyttäjän harteille.

Tässä luvussa perehdytään tarkemmin olioiden elinkaareen liittyviin kysymyksiin ja katsotaan, miten siihen liittyvät ongelmat on ratkaistu eri oliokielissä. Erityisesti tutkitaan, millaista tukea C++ antaa olioiden elinkaaren hallintaan.

3.1 Olion syntymä

Olioiden luominen saattaa vaatia monimutkaisiakin toimenpiteitä, ennen kuin olio on käyttövalmis. Tyypillisiä tällaisia toimenpiteitä ovat

- muistin varaaminen (merkkijono-olio varaa syntyessään tietyn minimimäärän muistia)
- toisten olioiden luominen (kirjasto-olio luo syntyessään tarvittavat kortisto-oliot)
- erinäiset rekisteröitymiset (palvelinolio rekisteröi syntyessään palvelunsa järjestelmänlaajuiseen tietokantaan)
- resurssien käyttöönotto (tietokantaolio avaa syntyessään tietokannan sisältävän tiedoston ja lukee sen muistiin)
- muut toimenpiteet (ikkunaolio piirtää syntyessään ikkunan ruudulle).

Nämä alkutoimenpiteet vastaavat muuttujien alustusta, mutta saattavat oliosta riippuen olla luonteeltaan paljon monimutkaisempia. Tietysti ei-olio-ohjelmoinnissakin on täytynyt tehdä vastaavia operaatioita (esim. tietorakenteen alustus), mutta niiden suorittaminen on yleensä jäänyt enemmän tai vähemmän ohjelmoijan itsensä vastuulle, ja yksi tyypillinen ohjelmointivirhe on ollut alustusfunktiokutsun unohtaminen. Olio-ohjelmoinnissa olisi luontevaa sysätä alustustoimenpiteet olion itsensä vastuulle, ja tähän eri oliokielet tarjoavat tukea vaihtelevassa määrin.

Tyypillisesti olion luomiseen liittyvät tehtävät jakautuvat oliokielissä kahteen vaiheeseen:

1. olion "datan" (siis jäsenmuuttujien) luominen. Jos jäsenmuuttujat ovat itse olioita, niiden luominen on puolestaan jälleen kaksivaiheinen prosessi. . .
2. muut alustustoimenpiteet.

Olion "datan" luominen muistuttaa niin paljon tavallisten muuttujien luomista, että käytännössä kaikki oliokielet tekevät sen automaattisesti olion luomisen yhteydessä. Tähän vaiheeseen kuuluu

muistin varaaminen olion jäsenmuuttujille ja mahdollisesti jäsenmuuttujien alustus. Muiden alustustoimenpiteiden "automatisoinnissa" oliokielissä on suuriakin eroja.

3.2 Olion kuolema

Kun oliota on käytetty ja siitä halutaan eroon, täytyy olion suorittaa yleensä luomiselleen vastakkaiset "siivoustoimenpiteet". Edellisen aliluvun listaa mukaillen tyypillisiä siivoustoimenpiteitä ovat muistinvapautus, toisten olioiden tuhoaminen, rekistereistä poistuminen, resurssien vapauttaminen ja vaikkapa ikkunan poistaminen ruudulta.

Ei-olio-ohjelmoinnissa tällaiset siivoustoimenpiteet ovat olleet muuttujan tai tietorakenteen käyttäjän vastuulla (ohjelmoijan on pitänyt muistaa kutsua siivous- tai vapautusfunktiota ennen muuttujan tuhoutumista). Samoin kuin olion luomisessakin, olion tuhoamiseen liittyvät toimenpiteet olisi kätevää saada olion itsensä vastuulle. Eri oliokielten tuki tähän vaihtelee aivan kuten olion luomisessakin.

3.3 Olion elinkaaren määräytyminen

Olion luomisen ja tuhoutumisen yhteydessä suoritettavien toimenpiteiden lisäksi oliokielissä on eroja sen suhteen, miten olion syntymä- ja varsinkin tuhoutumishetki ja näihin liittyvät toimenpiteeet määräytyvät. Tässä oliokielet voi jakaa karkeasti kolmeen ryhmään:

1. Kieli tarjoaa vain muistin varaamiseen ja vapauttamiseen tarvittavat operaatiot, ja olion alustaminen ja siivoaminen jäävät ohjelmoijan itsensä vastuulle. Muistin vapauttaminen tapahtuu jossain kielissä automaattisesti. Usein näissä kielissä on **roskienkeruu** (*garbage collection*): ohjelmassa on mukana kääntäjän tuottama koodi, joka osaa etsiä turhaksi jääneet muistialueet ja vapauttaa ne.

2. Kieli hoitaa olion tuhoamiseen liittyvät toimenpiteet automaattisesti, mutta olion tuhoutumishetki ei ole määrätty, eli se ei ole ohjelmoijan kontrolloitavissa. Olion luomisessa ei yleensä ole tätä vaihtoehtoa, koska useimmissa kielissä olion luominen on (ainakin lähes) aina ohjelmoijan itsensä päätettävissä.

3. Kieli pitää huolen alustustoimenpiteistä oliota luotaessa ja siivoustoimenpiteistä oliota tuhottaessa. Lisäksi olion luomisen ja tuhoamisen tapahtumishetki on tarkkaan määrätty.

Oliokielet sisältävät em. ominaisuuksia vaihtelevissa määrin ja eri tavoin sekoitettuna. Seuraavassa on esitelty joidenkin yleisimpien oliokielten tapoja toteuttaa asiat.

3.3.1 Modula-3

Modula-3 on oliokieli, joka kuuluu miltei puhtaasti kategoriaan 1. Kieli antaa mahdollisuuden varata oliolle muistia ja tässä yhteydessä antaa halutuille jäsenmuuttujille alkuarvot. Jäsenmuuttujien alustuksen lisäksi kieli ei kuitenkaan tue automaattisesti mitään muita alustustoimenpiteitä, vaan ohjelmoija voi halutessaan itse määritellä alustuksen suorittavan jäsenfunktion ja kutsua sitä omassa koodissaan heti olion luomisen jälkeen.

Samoin olion siivoustoimenpiteet jäävät ohjelmoijan itsensä vastuulle. Modula-3:ssa on roskienkeruu, joten olion varaaman muistin vapauttaminen tapahtuu automaattisesti. Roskienkeruu ei kuitenkaan suorita mitään oliokohtaisia siivoustoimenpiteitä, eli tässä mielessä Modula-3 vapauttaa automaattisesti vain olion alla olevan muistin, kun taas itse olio vain “unohdetaan”. Tämän vuoksi ohjelmoijan täytyy itse halutessaan määritellä siivousjäsenfunktio ja kutsua sitä sopivassa kohdassa ohjelmaa, kun oliota ei enää tarvita.

Modula-3:n käyttämä elinkaarimalli tuo mukanaan paljon vaaroja. Mikäli olio ei vaadi mitään erityisiä alustus- ja siivoustoimenpiteitä, riittää Modula-3:n tarjoama tuki elinkaarelle täysin. Mikäli tällaisia toimenpiteitä kuitenkin tarvitaan, ne jäävät kokonaan ohjelmoijan vastuulle, jolloin niiden unohtumisen vaara on suuri. Tällöin Modula-3:n roskienkeruun tuoma hyöty eliminoituu kokonaan, koska ohjelmoijan täytyy itse pitää kirjaa siitä, milloin oliota ei enää tarvita, ja kutsua kirjoittamaansa siivousjäsenfunktiota.

Listaus 3.1 seuraavalla sivulla näyttää Modula-3:lla kirjoitetun funktion `naytaViesti`, jossa luodaan ja tuhotaan olio. Tässä olion elinkaari kestää siis funktion suorituksen ajan.

```
................................ OkDialogi.i3 ................................
 1 INTERFACE OkDialogi;
 2 TYPE
 3   T <: Public;
 4   Public = OBJECT
 5            METHODS (* Esitellään olion julkinen rajapinta *)
 6                 alusta( viesti : TEXT );
 7                 siivoa();
 8                 odotaOKta();
 9            END; (* Public *)
10 (* Moduulin proseduuri, joka tulostaa viestin ja odottaa OK-nappia *)
11 PROCEDURE naytaViesti();
12 END OkDialogi.
................................ OkDialogi.m3 ................................
 1 MODULE OkDialogi;
 2 REVEAL
 3   T = Public BRANDED OBJECT
 4         (* Tähän paikalliset muuttujat *)
 5       OVERRIDES (* Määritellään olion tarjoamat palvelut *)
 6         alusta := Alustus;
 7         siivoa := Siivous;
 8         odotaOKta := OKodotus;
 9       END; (* T *)
10
11 PROCEDURE Alustus( self : T;        viesti : TEXT ) =
12 BEGIN (* Tänne ikkunan avaaminen, viestin kirjoittaminen siihen
13             ja muut luomistoimenpiteet *)
14 END Alustus;
15
16 PROCEDURE Siivous( self : T ) =
17 BEGIN
18   (* Tänne ikkunan sulkeminen ja muut siivoustoimenpiteet *)
19 END Siivous;
20
21 PROCEDURE OKodotus( self : T ) =
22 BEGIN
23   (* Tänne nappulan painamisen odottamisen koodi *)
24 END OKodotus;
25
26 PROCEDURE naytaViesti() =
27 VAR dialogi := NEW(T); (* Varataan oliolle muistialue *)
28 BEGIN
29   dialogi.alusta("Virhe!"); (* Alustetaan olio *)
30   dialogi.odotaOKta();      (* Odota, että käyttäjä kuittaa viestin *)
31   dialogi.siivoa();         (* Olion siivoustoimenpiteet *)
32   (* proseduurista palattaessa roskienkeruu pitää huolen olion muistista *)
33 END naytaViesti;
34
35 BEGIN
36   (* Moduulin alustuskoodi *)
37 END OkDialogi.
```

LISTAUS 3.1: Esimerkki olion elinkaaresta Modula-3:lla

3.3.2 Smalltalk

Smalltalk-kielessä olioiden elinkaari on olioiden tuhoutumisen kannalta samanlainen kuin Modula-3:ssa eli olioille ei suoriteta automaattisesti mitään siivoustoimenpiteitä ja roskienkeruu pitää huolen olion varaaman muistin vapauttamisesta. Sen sijaan olioiden luomisen yhteydessä Smalltalk antaa mahdollisuuden jonkinasteiseen automatisointiin.

Smalltalkissa olioita luodaan antamalla luokalle käsky `new`. Tämä puolestaan luo uuden olion ja palauttaa sen paluuarvonaan. Luomiskäskylle voi antaa parametreja ja sen sisältämä koodi voi suorittaa olion alustustoimenpiteitä, joten ohjelmoijan ei tarvitse itse muistaa kutsua alustusjäsenfunktiota luomisen yhteydessä.

Listaus 3.2 seuraavalla sivulla sisältää esimerkin olion elinkaaresta Smalltalkilla kirjoitettuna (Smalltalkissa ei oikeastaan ole “ohjelmalistauksen” käsitettä, listauksessa on vain kirjoitettu tarvittavat koodilohkot peräkkäin).

3.3.3 Java

Java-kielessä olioiden elinkaari on hieman kaksipiippuinen. Olioita luotaessa Java tarjoaa mahdollisuuden alustustoimenpiteisiin **rakentajaksi** kutsutun jäsenfunktion avulla. Tätä jäsenfunktiota kutsutaan automaattisesti oliota luotaessa. Sen sijaan olion siivoustoimenpiteet ovat ongelmallisempia.

Myös Javassa on roskienkeruu, eli käytöstä poistuneiden olioiden viemä muistitila ei tuota ongelmia. Javassa ohjelmoijalla ei ole mitään mahdollisuutta ilmoittaa, milloin olion elinkaari loppuu, vaan roskienkeruualgoritmi päättelee itse, milloin olio on tarpeeton eli milloin siihen ei ole ulkopuolisia viitteitä. Kieli ei kuitenkaan määrittele, kuinka pian olion tarpeettomaksi tulon jälkeen roskienkeruu suoritetaan.

Periaatteessa Java antaa ohjelmoijalle mahdollisuuden kirjoittaa ns. finalize-jäsenfunktioita, joita roskienkeruu kutsuu ennen kuin olio siivotaan muistista. Tämä mekanismi ei kuitenkaan ole aina käyttökelpoinen, koska ohjelmoija ei voi tietää, milloin roskienkeruu sattuu finalize-jäsenfunktiota kutsumaan. On myös mahdollista, että ohjelman suoritus loppuu ennen kuin roskienkeruu siivoaa olion muistista, jolloin finalizea ei kutsuta ollenkaan.

```
 1 Object subclass: #OkDialogi
 2   instanceVariableNames: 'viesti '
 3   classVariableNames: ''
 4   poolDictionaries: ''
 5   category: 'Oliokirja'
 6
 7 alusta: aViesti
 8   "OkDialogi-olion luomistoimenpiteet"
 9   viesti := aViesti.
10   "Tänne ikkunan avaaminen, viestin kirjoittaminen siihen ja muut
11   luomistoimenpiteet"
12   ^ self
13
14 new: aViesti
15   "OkDialogi-olion alusta-jäsenfunktion kutsumiseen tarvittava kikka"
16   ^ super new alusta: aViesti
17
18 odotaOKta
19   "Täällä odotetaan OK-napin painamista"
20
21 siivoa
22   "OkDialogi-olion siivoustoimenpiteet"
23   "Tänne ikkunan sulkeminen ja muut siivoustoimenpiteet"
24
25 naytaViesti
26   "Funktio joka tulostaa viestin ja odottaa OK-nappia"
27   | dialogi | "Paikallinen muuttuja"
28   dialogi := OkDialogi new: 'Virhe!!'. "Olio syntyy, luomistoimenpiteet"
29   dialogi odotaOKta.                   "Odota, että käyttäjä kuittaa viestin"
30   dialogi siivoa                       "Suorita siivoustoimenpiteet"
31   "Funktiosta palattaessa roskienkeruu pitää huolen olion muistista"
```

LISTAUS 3.2: Esimerkki olion elinkaaresta Smalltalkilla

Näin ollen Javassa olion alustustoimenpiteet suoritetaan automaattisesti, mutta jos siivoustoimenpiteet vaativat muistin vapauttamista monimutkaisempia asioita, ohjelmoija joutuu käytännössä koodaamaan oman siivousjäsenfunktionsa ja kutsumaan sitä itse sellaisessa ohjelman kohdassa, jossa tietää olion tulleen tarpeettomaksi. Listaus 3.3 seuraavalla sivulla sisältää esimerkin olion elinkaaresta Javalla kirjoitettuna.

```
 1 public class OkDialogi // OkDialogi-luokan esittely
 2 {
 3   public OkDialogi(String viesti)
 4   { // Tänne ikkunan avaaminen, viestin kirjoittaminen siihen
 5     // ja muut luomistoimenpiteet
 6   }
 7
 8   public void siivoa()
 9   { // Tänne ikkunan sulkeminen ja muut siivoustoimenpiteet
10   }
11
12   public void odotaOKta()
13   { // Tähän nappulan painamisen odottamisen koodi
14   }
15
16   // Tähän luokan loput jäsenfunktiot ja sisäinen toteutus
17
18   // Luokkafunktio joka tulostaa viestin ja odottaa OK-nappia
19   public static void naytaViesti()
20   {
21     // Olio luodaan, luomistoimenpiteet
22     OkDialogi dialogi = new OkDialogi("Virhe!");
23     dialogi.odotaOKta();              // Odota, että käyttäjä kuittaa viestin
24     dialogi.siivoa();                 // Olion siivoustoimenpiteet
25     // Funktiosta palattaessa roskienkeruu pitää huolen olion muistista
26   }
27 }
```

LISTAUS 3.3: Esimerkki olion elinkaaresta Javalla

3.3.4 C++

C++ sisältää vaihtelevantasoisen tuen olioiden elinkaaren hallintaan. Tietyissä tapauksissa oliot tuhoutuvat automaattisesti, toisissa tuhoaminen taas jätetään ohjelmoijan vastuulle. Kielessä voi Javan tapaan määritellä olion alustustoimenpiteet suorittavan rakentajajäsenfunktion, jota kutsutaan automaattisesti oliota luotaessa. Vastaavasti C++:ssa on mahdollista kirjoittaa **purkaja**, joka on jäsenfunktio, joka sisältää kaikki siivoustoimenpiteet ja jota C++ kutsuu automaattisesti oliota tuhottaessa. C++:n rakentajista ja purkajista kerrotaan enemmän aliluvussa 3.4.

Staattinen elinkaari

C++:ssa on kahdentyyppisiä olioiden elinkaaria. Olioilla, jotka luodaan tavallisen muuttujan tapaan esimerkiksi funktion paikallisiksi muuttujiksi, globaaleiksi muuttujiksi[♈] tai olion jäsenmuuttujiksi, on **staattinen elinkaari**. Tämä tarkoittaa, että olion syntymä- ja tuhoutumishetki on määrätty jo käännösaikana ja kääntäjä osaa automaattisesti suorittaa tarvittavat luomistoimenpiteet olion syntyessä ja vastaavasti siivoustoimenpiteet heti olion tuhoutuessa.

Listaus 3.4 seuraavalla sivulla sisältää esimerkin olion elinkaaresta C++:lla kirjoitettuna. Riveillä 20–26 on esimerkki staattisen elinkaaren käyttämisestä. `OkDialogi`-olio `dialogi` luodaan aivan kuten tavallinen paikallinen muuttuja, ja sen rakentajalle annetaan luomisen yhteydessä tarvittavat parametrit, tässä tapauksessa dialogin teksti.

Koska `dialogi` on funktion paikallinen muuttuja, sen elinkaari rajautuu funktion sisälle. Funktion lopussa rivillä 26 olio tuhotaan automaattisesti ja sen purkajaa kutsutaan siivoustoimenpiteitä varten. Jos tällaisia paikallisia olioita on useita, ne tuhotaan käänteisessä järjestyksessä niiden luomisjärjestykseen nähden.

Staattisen elinkaaren käyttäminen on C++:lla ohjelmoitaessa suositeltavaa aina, kun se on mahdollista, koska tällöin ohjelma pitää automaattisesti huolen olioiden tuhoamisesta elinkaaren lopussa. Tällöin ei ole vaaraa muistivuodoista eikä siivoustoimenpiteiden unohtumisesta.

Paikallisten muuttujien, globaalien muuttujien ja olioiden jäsenmuuttujien lisäksi funktioiden parametreilla, luokkien luokkamuuttujilla ja kääntäjän luomilla väliaikaisolioilla on C++:ssa staattinen elinkaari. Ohjelmoijan ei itse tarvitse — eikä hän voikaan — huolehtia tällaisten olioiden tuhoamisesta, vaan kääntäjä itse tuhoaa oliot niiden elinkaaren päättyessä.

Staattinen elinkari helpottaa ohjelmoijan työtä, kun kääntäjä pitää huolen olioiden tuhoamisesta. Tästä huolimatta staattisenkaan elinkaaren käyttäminen ei saisi tuudittaa väärään turvallisuuden tunteeseen. C++:ssa on mahdollista tehdä vakavia ohjelmointivirheitä staattisen elinkaarenkin avulla. Jos funktio esimerkiksi palauttaa paluuarvonaan osoittimen (tai viitteen) paikalliseen muuttujaan, ehtii tämä muuttuja tuhoutua ennen kuin kutsuja pystyy käyttämään osoitinta. Tuloksena on viallinen osoitin, jonka läpi viittaamisen vaikutuk-

[♈] Ethän käytä globaaleja muuttujia, ethän?☹

```
1  // OkDialogi-luokan esittely
2  class OkDialogi
3  {
4  public:
5     OkDialogi(string const& viesti);
6     ~OkDialogi();
7     void odotaOKta();
8     // Tähän luokan loput jäsenfunktiot ja sisäinen toteutus
9  };
10 // OkDialogi-olion luomistoimenpiteet
11 OkDialogi::OkDialogi(string const& viesti)
12 { // Tänne ikkunan avaaminen, viestin kirjoittaminen siihen
13    // ja muut luomistoimenpiteet
14 }
15 // OkDialogi-olion siivoustoimenpiteet
16 OkDialogi::~OkDialogi()
17 { // Tänne ikkunan sulkeminen ja muut siivoustoimenpiteet
18 }
19 // Funktio joka tulostaa viestin ja odottaa OK-nappia

20 void naytaViesti1() // Staattisella dialogin elinkaarella
21 {
22    OkDialogi dialogi("Virhe!"); // Olio syntyy, luomistoimenpiteet
23    dialogi.odotaOKta();          // Odota, että käyttäjä kuittaa viestin
24    // Funktion loppuessa dialogin elinkaari päättyy, siivoustoimenpiteet
25    // suoritetaan ja olio tuhoutuu
26 }
27 // Dynaamisella dialogin elinkaarella (typerää)
28 void naytaViesti2()
29 {
30    OkDialogi* dialogip = new OkDialogi("Virhe!"); // Luomistoimenpiteet
31    dialogip->odotaOKta();          // Odota, että käyttäjä kuittaa viestin
32    delete dialogip; dialogip = 0; // Tuhoaminen ja siivoustoimenpiteet
33 }
```

LISTAUS 3.4: Esimerkki olion elinkaaresta C++:lla

set ovat täysin määrittelemättömät. Sen vuoksi staattisen elinkaaren käytössä onkin tärkeää, että ohjelmoija todella tiedostaa sen, että olio tuhoutuu automaattisesti tietyssä kohtaa ohjelmaa.

Dynaaminen elinkaari

Toinen mahdollisuus on jättää olion elinkaaresta huolehtiminen ohjelmoijan itsensä vastuulle. Tällöin olio ei automaattisesti tuhoudu ollenkaan, vaan se täytyy erikseen tuhota. Tällainen on usein tarpeen, jos olio luodaan yhdessä funktiossa ja tuhotaan toisessa. Myös dynaaminen sitominen ja polymorfismi (aliluku 6.5) vaativat usein dynaamisen elinkaaren käyttöä.

Listauksen 3.4 riveillä 27–33 oleva funktio käyttää dialogissa dynaamista elinkaarta. Olion elinkaari saadaan dynaamiseksi luomalla olio **`new`**-operaattorilla. Parametreikseen **`new`** ottaa luotavan olion luokan sekä rakentajan parametrit. Tämän jälkeen se luo uuden olion, suorittaa alustustoimenpiteet ja palauttaa osoittimen olioon. Oliota käytetään tämän jälkeen ko. osoittimen läpi. Kun oliosta halutaan päästä eroon, täytyy se erikseen tuhota osoittimen päästä operaattorilla **`delete`**, joka suorittaa olion siivoustoimenpiteet ja tuhoaa olion. Näitä operaattoreita käsitellään tarkemmin aliluvussa 3.5.

Dynaamisen elinkaaren käyttö on “vaarallisempaa” kuin staattisen, koska ohjelmoijan täytyy itse pitää huoli siitä, että kaikki oliot tuhotaan, kun niitä ei enää tarvita. Erityisen vaikeaa tämä on virhetilanteiden sattuessa, jolloin on vaikeaa muistaa, mitkä oliot on jo luotu ja täytyy näin ollen tuhota. Varsinkin poikkeusmekanismia (luku 11) käytettäessä tämä on vaikeaa. C++-standardi tarjoaa `auto_ptr`-tyypin, joka helpottaa jonkin verran dynaamisen elinkaaren olioiden hallintaa. Siitä kerrotaan aliluvussa 11.7.

3.4 C++: Rakentajat ja purkajat

Kuten edellä on käynyt ilmi, olioiden luominen ja tuhoaminen eroaa tavallisten muuttujien luomisesta ja tuhoamisesta siinä, että olioiden luominen ja tuhoaminen saattaa vaatia alustus- ja siivoustoimenpiteitä luokasta riippuen. C++:ssa tämä on toteutettu niin, että joka luokalla on kaksi erityisjäsenfunktiota, **rakentaja** ja **purkaja**, joita kutsutaan automaattisesti aina olion luomisen ja tuhoamisen yhteydessä.

3.4.1 Rakentajat

Rakentaja (*constructor*) on jäsenfunktio, jonka tehtävänä on hoitaa kaikki uuden olion alustamiseen liittyvät toimenpiteet. Siitä käytetään joskus myös nimitystä "muodostin". Rakentajan suorittamia toimenpiteitä ovat ainakin *jäsenmuuttujien alustaminen*, olioon kuuluvien olion ulkopuolisten tietorakenteiden ja olioiden luominen sekä mahdollisesti olion rekisteröiminen jonnekin yms.

Kääntäjä tunnistaa rakentajan sen nimestä: rakentajan nimi on aina sama kuin luokan nimi. Rakentaja voi saada valinnaisen määrän parametreja, mikäli niitä tarvitaan olion alustamiseen. Listaus 3.5 näyttää `Paivays`-luokan rakentajan, joka saa parametreinaan uuden päiväyksen päivän, kuukauden ja vuoden. Rakentaja ei koskaan palauta mitään paluuarvoa, ja tästä johtuen rakentajan esittelyssä ja määrittelyssä ei paluutyyppiä merkitä lainkaan — ei edes avainsanalla **`void`**.

Rakentajan esittely luokan esittelyn yhteydessä ei eroa tavallisen jäsenfunktion esittelystä muuten kuin paluutyypin puuttumisen osalta. Sen sijaan sen määrittely on hieman tavallisuudesta poikkea-

paivays.hh

```
1  class Paivays
2  {
3  public:

4    Paivays(unsigned int p, unsigned int k, unsigned int v);
         ⋮
22 };
```

paivays.cc

```
6  Paivays::Paivays(unsigned int p, unsigned int k, unsigned int v)
7    : paiva_(p), kuukausi_(k), vuosi_(v)
8  { // Ei mitään tehtävää täällä
9  }
```

LISTAUS 3.5: `Paivays`-luokan rakentaja

va. Rakentajan määrittely on muotoa

```
Luokannimi::Luokannimi(Parametrilista)
  : jmuutt1(alkuarvo1), jmuutt2(alkuarvo2), ... // Alustuslista
{
  // Muut alustustoimenpiteet
}
```

Alustuslista (*initialization list*) sisältää luettelon luokan jäsenmuuttujista ja jokaisen jäsenmuuttujan perässä suluissa jäsenmuuttujan alustamiseen tarvittavat alkuarvot. Joissain C++-oppikirjoissa alustuslistaa ei käytetä lainkaan, vaan jäsenmuuttujiin sijoitetaan arvot rakentajan koodilohkossa. **Tämä ei ole hyvää tyyliä** (sijoituksen ja alustuksessa käytettävän kopioinnin eroja käsitellään myöhemmin aliluvussa 7.2.1).

Rakentajan runkoon (siis aaltosulkujen sisälle) voi kirjoittaa muuta olion alustamiseen tarvittavaa koodia. Varsin usein käy kuitenkin niin, että runko jää tyhjäksi, koska luokan oliot eivät tarvitse muita alustustoimenpiteitä kuin jäsenmuuttujien alustamisen.

Luokalla voi olla useita vaihtoehtoisia rakentajia, kunhan ne vain saavat eri määrän tai eri tyyppisiä parametreja niin, että olion luomisen yhteydessä kääntäjä voi parametrien perusteella päätellä, mitä rakentajaa tulee kutsua.[♉]

Jäsenmuuttujien alustaminen

Mikäli jäsenmuuttuja on jotakin perustyyppiä, kuten **`int`**, annetaan sille alustuslistassa yksinkertaisesti alkuarvo kuten listauksen 3.5 esimerkissä. Jos taas jäsenmuuttuja on olio, annetaan alustuslistassa jäsenmuuttujan nimen perään sen rakentajan tarvitsemat parametrit — aivan kuten normaalissa olion määrittelyssä olion nimen perään tulevat suluissa rakentajan parametrit. Listaus 3.6 seuraavalla sivulla näyttää yksinkertaisen `Henkilo`-luokan rakentajan, jossa alustetaan `Paivays`-tyyppinen jäsenmuuttujaolio `syntymapvm_`.

[♉] Tätä kielen ominaisuutta, jossa ohjelmassa voi olla useita samannimisiä funktioita, jotka eroavat toisistaan parametrien määrän tai tyyppien perusteella, kutsutaan funktioiden **kuormittamiseksi** (*overloading*). C++:ssa lähes mitä tahansa funktioita voi kuormittaa, kunhan vain funktiokutsun yhteydessä kääntäjä pystyy parametreista päättelemään oikean version funktiosta.

```
1  class Henkilo
2  {
3  public:
4    Henkilo(string const& nimi, unsigned int syntymavuosi,
5            unsigned int syntymakk, unsigned int syntymapv);
       ⋮
6  private:
7    string nimi_;
8    Paivays syntymapvm_;
       ⋮
9  };
       ⋮
10 Henkilo::Henkilo(string const& nimi, unsigned int syntymavuosi,
11                  unsigned int syntymakk, unsigned int syntymapv)
12   : nimi_(nimi), syntymapvm_(syntymapv, syntymakk, syntymavuosi)
13 {
14   // Tänne loput henkilön alustuksesta
15 }
```

LISTAUS 3.6: Esimerkki rakentajasta olion ollessa jäsenmuuttujana

Oletusrakentaja

Oletusrakentajaksi (*default constructor*) kutsutaan rakentajaa, joka ei saa yhtään parametria. Sen täytyy siis pystyä alustamaan olio ilman ulkopuolista tietoa, joten on kätevää ajatella, että sen avulla luodaan "oletusarvoinen" olio. Oletusrakentajaa kutsutaan seuraavissa tapauksissa:

- Jos olio luodaan ilman että sen rakentajalle annetaan parametreja. Huomaa, että tällöin myös parametrien ympärillä normaalisti olevat sulut jäävät pois:

  ```
  Lista l; // Luodaan tyhjä lista, kutsutaan oletusrakentajaa
  ```

- Jos jäsenmuuttujana olevan olion alustus unohtuu "isäntäolion" rakentajan alustuslistasta, alustetaan jäsenmuuttujaolio käyttäen oletusrakentajaa, mikäli sellainen on olemassa (muuten annetaan virheilmoitus). Se, että alustuksen unohtamisesta ei tule virheilmoitusta, on yksi syy välttää oletusrakentajia.

Perustyyppiä olevilla muuttujilla ei ole oletusrakentajaa, joten tyyliin `int i;` esitellyt muuttujat jäävät alustamatta, kuten C-kielessäkin. Samoin käy perustyyppisille jäsenmuuttujille, joita ei alusteta rakentajassa. Tämän vuoksi on erittäin tärkeää, että kaikki perustyyppiä olevat (jäsen)muuttujat alustetaan aina johonkin järkevään alkuarvoon!

Oletusrakentajan toinen erityisominaisuus on, että jos luokalle ei ole kirjoitettu *yhtäkään* rakentajaa, tekee kääntäjä sille automaattisesti tyhjän oletusrakentajan, joka ei tee mitään muuta kuin alustaa kaikki jäsenmuuttujat niiden oletusrakentajilla (jos tämä ei onnistu, annetaan virheilmoitus). Koska kääntäjän itse tekemät rakentajat harvoin tekevät sitä mitä halutaan, kannattaa jokaiseen luokkaan kirjoittaa aina oma rakentaja.

Oletusrakentajaa käytetään myös silloin, kun luodaan C++:n perustaulukko, joka sisältää olioita:

```
Lista listaTaulukko[10]; // Luodaan 10 tyhjän listan taulukko
```

Tällöin jokainen taulukon alkio alustetaan käyttäen oletusrakentajaa. Mikäli luokalla ei ole oletusrakentajaa, ei siitä voi myöskään tehdä tällaisia taulukoita. Tämä ongelma ratkeaa helpoiten käyttämällä STL:n `vector`-luokkaa, johon oikein alustetut oliot voi lisätä yksi kerrallaan vaikkapa silmukassa (`vector`-luokasta on lyhyt esittely liitteen A aliluvussa A.3).

Kopiorakentaja

Kopiorakentaja (*copy constructor*) on rakentaja, joka saa parametrinaan viitteen jo olemassa olevaan saman luokan olioon. Sen tehtävänä on luoda identtinen kopio parametrina saadusta oliosta, ja kääntäjä kutsuu sitä automaattisesti tietyissä tilanteissa, joissa kopion luominen on tarpeen. Kopiorakentajaa käsitellään tarkemmin aliluvussa 7.1.2.

3.4.2 Purkajat

Purkajaksi (*destructor*) kutsutun jäsenfunktion tehtävänä on suorittaa tarvittavat siivoustoimenpiteet olion tuhoutuessa. Tällaisia ovat

muiden muassa olioon kuuluvien olion ulkopuolisten tietorakenteiden ja olioiden tuhoaminen sekä tiedostojen sulkeminen. Purkajasta käytetään myös nimityksiä "hajotin" ja "hävitin".

Kuten rakentajankin tapauksessa, kääntäjä tunnistaa purkajan sen nimestä. Purkajan nimi alkaa matomerkillä '∼', jonka perään tulee heti luokan nimi.[♊] Purkaja ei saa parametreja, eikä sen esittelyyn eikä määrittelyyn merkitä paluuarvoa kuten ei rakentajiinkaan. Listaus 3.7 näyttää päiväysluokan purkajan esimerkkinä purkajan syntaksista.

Kun olion tuhoutumisen aika tulee, kääntäjä pitää itse automaattisesti huolen olion jäsenmuuttujien tuhoamisesta. Näin ohjelmoijan ei tarvitse purkajaa kirjoittaessaan huolehtia tästä. Olion jäsenmuuttujat tuhotaan vasta purkajan varsinaisen koodin suorituksen jälkeen, joten itse purkajan koodissa jäsenmuuttujiin voi viitata normaalisti. On kuitenkin huomattava, että automaattinen tuhoaminen koskee vain olion omia jäsenmuuttujia, ei esimerkiksi dynaamisesti osoittimien päähän luotuja olioita.

Purkajien toiminta on muutenkin varsin automaattista. Ohjelmoijan ei koskaan tarvitse itse kutsua luokan purkajaa, vaan kääntäjä pitää huolen purkajan kutsumisesta, kun olio tuhotaan. Tämä tapahtuu mm. seuraavissa tapauksissa:

[♊] Lausekkeessa esiintyessäänhän matomerkki '∼' on C++:ssä "bitti-not"-operaattori, jolla käännetään luvun bitit päinvastaisiksi. Niinpä purkajan nimi ikään kuin kuvaa sitä, että se on rakentajalle vastakkainen operaatio...

paivays.hh

```
 1 class Paivays
 2 {
 3 public:
       ⋮
 5   ~Paivays();
       ⋮
22 };
```

paivays.cc

```
11 Paivays::~Paivays()
12 { // Ei siivottavaa
13 }
```

LISTAUS 3.7: Paivays-luokan purkaja

- Paikallisten olioiden purkajia kutsutaan automaattisesti juuri ennen kuin niiden elinkaari (eli näkyvyysalue) loppuu koodilohkon loppuaaltosulun kohdalla. Huomaa, että tämä tarkoittaa sitä, että paikalliset oliot tuhotaan vasta **`return`**-lauseen *jälkeen*.
- Globaalien olioiden purkajia kutsutaan `main`-funktion loputtua.
- Funktion parametrioliot tuhoutuvat aivan kuten funktion paikalliset oliot.
- Jäsenmuuttujina olevien olioiden purkajia kutsutaan, kun niiden "isäntäolio" tuhoutuu.
- Dynaamisesti luotujen olioiden purkajia (katso seuraava aliluku) kutsutaan, kun oliota tuhotaan **`delete`**llä.
- Taulukossa olevien olioiden purkajia kutsutaan, kun taulukko tuhoutuu.

Mikäli olion tuhoutumisessa ei tarvita mitään erityisiä siivoustoimenpiteitä, luokalle ei periaatteessa ole pakko kirjoittaa purkajaa. Tällaisessakin tapauksessa kannattaa kuitenkin kirjoittaa luokalle tyhjä purkaja, koska muutoin koodin lukijalla herää helposti epäilys, että luokan kirjoittaja on vain unohtanut kirjoittaa purkajan.

3.5 C++: Dynaaminen luominen ja tuhoaminen

Kuten aiemmin tässä luvussa on jo todettu, staattisen elinkaaren käyttö C++-ohjelmoinnissa on suotavaa, koska tällöin kääntäjä pitää huolen siitä, että kaikki luodut oliot aikanaan myös tuhotaan. Varsin usein tulee kuitenkin tarve luoda olioita, joiden elinkaarta ei voida käännösaikana rajata tiettyyn osaan koodia. Tällaiset oliot täytyy luoda dynaamisella elinkaarella, jolloin kaikki vastuu olioiden tuhoamisesta siirtyy ohjelmoijalle. Tässä luvussa "olion luomista dynaamisella elinkaarella" kutsutaan yksinkertaisesti "olion dynaamiseksi luomiseksi".

Muistivuodot ovat jo pitkään olleet tyypillisiä ohjelmissa, joissa ohjelmoija on unohtanut vapauttaa dynaamisesti varaamansa muistin. Usein muistivuotoihin kuitenkin suhtaudutaan vähättelevästi, koska "käyttöjärjestelmä kuitenkin vapauttaa muistin ohjelman lopuessa". Olio-ohjelmoinnissa tilanne on kuitenkin vakavampi, koska

oliolla on yleensä purkaja, jonka koodi suoritetaan olion tuhoamisen yhteydessä. Mikäli nyt ohjelmoija unohtaa tuhota dynaamisesti varatun olion, *ei purkajaa koskaan suoriteta*! Nyt siis pelkän muistivuodon lisäksi myös osa ohjelmakoodista jää suorittamatta — mahdollisesti vakavin seurauksin.

Esimerkiksi kelpaa mainiosti tietokantaolio, joka säilyttää tietokantaa tiedostossa. On hyvin mahdollista, että olion koodi on kirjoitettu optimoidusti niin, että olio puskuroi tietoa muistiin ja päivittää tiedostossa olevaa tietokantaa vain tietyin väliajoin. Jos nyt tällainen tietokantaolio on luotu dynaamisesti ja se unohdetaan tuhota, sen purkaja jää suorittamatta ja muistiin puskuroitu tieto päivittämättä tiedostoon.

Olioiden dynaamiseen luomiseen liittyy muitakin ongelmia. Niistä yleisimpiä ovat

- olion tuhoamisen unohtaminen
- muistin loppuminen (olion luomisen epäonnistuminen)
- olion tuhoaminen kahteen kertaan
- jo tuhotun olion käyttäminen (tämä vaara on muuallakin kuin dynaamisen elinkaaren yhteydessä)
- sekoilut C++:n taulukkojen tuhoamisessa (aliluku 3.5.3).

Kaikista näistä vaaroista huolimatta olioiden dynaaminen luominen on varsin usein tarpeellista C++:lla ohjelmoitaessa. Sitä käytettäessä on vain oltava erittäin huolellinen.

3.5.1 new

Olion luominen dynaamisesti tapahtuu syntaksilla

```
new Tyyppinimi(alustusparametrit)
```

Operaattori **new** luo uuden (nimettömän) olion ja palauttaa osoittimen siihen. Esimerkiksi uuden päiväysolion ja uuden kokonaisluvun voi luoda seuraavasti:

```
Paivays* pvm_p = new Paivays(1,1,1000);
int* luku_p = new int(5); // Uusi int, alkuarvona 5
```

Mikäli uudelle oliolle ei saada varatuksi riittävästi muistia, heittää **new** poikkeuksen, jonka tyyppi on `std::bad_alloc`. Tämän poikkeuksen sieppaaminen vaatii otsikkotiedoston <**new**> lukemista.

.... ***Ekskursio:*** *Poikkeukset (lisää luvussa 11)* ....

Poikkeukset (*exception*) ovat C++:ssa uusi tapa hoitaa ohjelman virhetilanteita. Kun ohjelmassa havaitaan virhe, virheen havainnut koodinkohta heittää (*throw*) "ilmaan" poikkeusolion, jonka ylemmillä ohjelman kutsutasoilla olevat virhekäsittelijät voivat siepata (*catch*).

Yksinkertaisimmillaan poikkeusmekanismi näyttää seuraavanlaiselta:

```
1 try
2 {
3   // Tänne koodi, jossa syntyviä virhetilanteita tarkkaillaan
4 }
5 catch (VirheenTyyppi& vo)
6 { // Virheolio vastaanotetaan "parametrina"
7   // Tänne virhekäsittelykoodi, joka voi käyttää virheoliota vo
8 }
9 // Täältä jatketaan
```

Kun tällainen rakenne tulee ohjelmassa vastaan, ohjelman suoritus siirtyy suoraan **try**-lohkon sisälle, jossa jatketaan normaalisti. Mikäli virhettä ei lohkon sisällä tapahdu, hypätään **catch**-lohkon yli ja jatketaan ohjelman suoritusta koko **try**-**catch**-rakenteen jälkeen.

Jos **try**-lohkon sisällä heitetään poikkeus, tarkastetaan kelpaako se tyyppinsä perusteella **catch**-lohkon parametriksi. Jos se kelpaa, siirtyy ohjelman suoritus **catch**-lohkon sisään. Kun tämän lohkon koodi on suoritettu, ohjelman suoritus jatkuu **try**-**catch**-rakenteen jälkeisestä ohjelmakoodista. Tämä tarkoittaa sitä, että **try**-lohkoon ei enää palata virhekäsittelyn jälkeen.

Virhekäsittelijä voi myös lopuksi heittää virheen uudelleen ilmaan komennolla **throw;**, jos virheestä ei voida toipua kokonaan eli virhe halutaan välittää vielä ylemmäs ohjelmassa. Jos poikkeusta ei oteta kiinni missään, keskeytetään ohjelman suoritus virheilmoitukseen.

Yllä oleva poikkeuskäsittelyn kuvaus on pahasti puutteellinen mutta riittää tässä vaiheessa.

....

Muistin loppumiseen tulisi aina varautua. Tyypillinen esimerkki dynaamisesta olion luomisesta ja muistin loppumisen hallinnasta löytyy riveiltä 12–21 listauksesta 3.8.

Joskus poikkeukset ovat turhan raskas tapa varautua muistin lop-

```
1  #include "paivays.hh"
2
3  #include <new>
4  #include <cstdlib>
5  #include <iostream>
6  using std::cout;
7  using std::cerr;
8  using std::endl;
9
10 int main()
11 {
12   Paivays* joulu_p = 0; // Nollataan kaiken varalta
13   try
14   {
15     joulu_p = new Paivays(24,12,1999); // Poikkeus, jos muisti loppuu
16   }
17   catch (std::bad_alloc&)
18   { // Virhekäsittely, jos muisti loppui
19     cerr << "Muisti loppui, lopetan!" << endl;
20     return EXIT_FAILURE;
21   }
22
23   Paivays* vappu_p = new(std::nothrow) Paivays(1,5,2000);
24   if (vappu_p == 0)
25   {
26     // Virhekäsittely, jos muisti loppui
27     cerr << "Muisti loppui, lopetan!" << endl;
28     delete joulu_p; joulu_p = 0; // Tuhotaan jo varattu olio!
29     return EXIT_FAILURE;
30   }
31
32   // Täältä jatketaan, jos virhettä ei sattunut
33   cout << "Joulun ja vapun välissä päiviä on "
34        << joulu_p->paljonkoEdella(*vappu_p) << endl;
35
36   delete vappu_p; vappu_p = 0; // Dynaamisesti luodut oliot tuhottava
37   delete joulu_p; joulu_p = 0;
38 }
```

LISTAUS 3.8: Esimerkki olion dynaamisesta luomisesta **new**'llä

pumiseen. Tällaisia tilanteita varten **new**-operaattorista on myös olemassa versio **new**(std::nothrow), joka muistin loppuessa ei heitä poikkeusta vaan palauttaa nollaosoittimen. Ko. operaattorin käyttö vaatii myös otsikkotiedoston <**new**> lukemista. Tietysti muistin loppumiseen täytyy varautua tässäkin tapauksessa. Listauksen 3.8 riveillä 23–30 on esimerkki tästä. Vanhoissa C++-kielen versioissa nollaosoittimen palauttaminen oli **new**'n oletustoiminto, koska poikkeuksia ei vielä ollut.

Yleensä olioiden dynaamiseen luomiseen kannattaa käyttää "normaalia" poikkeuksen heittävää **new**'tä, koska tällöin ohjelman suoritus ainakin keskeytyy, jos muistin loppumiseen ei varauduta. Nollaosoittimen palauttava nothrow-versio sen sijaan jatkaa ohjelman suoritusta, kunnes ohjelma jossain vaiheessa todennäköisesti sekoaa, kun se yrittää käyttää olematonta oliota. Tässä vaiheessa saattaa kuitenkin olla todella vaikea paikallistaa, mistä virhe oikeastaan on johtunut.

3.5.2 delete

Ohjelmoijan täytyy itse muistaa tuhota **new**'llä luomansa oliot. Tämä tapahtuu operaattorilla **delete**, jolle annetaan operandina osoitin **new**'llä varattuun olioon. Operaattori **delete** tuhoaa kyseisen olion ja vapauttaa sen käyttämän muistitilan. Tuhoamisen yhteydessä olion purkajaa kutsutaan normaalisti (periytymistä käytettäessä luokan purkajan tulee olla virtuaalinen, katso aliluku 6.5.5). Jos **delete**lle antaa olioon osoittavan osoittimen sijaan nollaosoittimen, ei kyseessä ole virhe. Tällöin **delete** ei vain tee mitään.

Koska olion tuhoamisen jälkeen **delete**lle annettu osoitin ei enää osoita mihinkään järkevään, kannattaa sen arvoksi asettaa nolla. Tämä ei ole pakollista, mutta kylläkin hyvän ohjelmointityylin mukaista.

Poikkeuksiin ei **delete**n tapauksessa tarvitse varautua, koska olion tuhoamisessa ei muistin loppumista tai muita virheitä pitäisi tapahtua. Poikkeuksista kertovassa luvussa 11 käsitellään mahdollisia olion tuhoamisen aikaisia virhetilanteita tarkemmin.

Esimerkkilistauksen 3.8 riveillä 36–37 tuhotaan dynaamisesti varatut oliot. Huomaa myös rivi 28, jossa virheenkäsittelyn yhteydessä täytyy muistaa tuhota dynaamisesti luotu olio joulu_p. Tällaiset virhetilanteissa tapahtuvat tuhoamiset unohtuvat erittäin helposti ja

ovat yksi lisäsyy valita staattinen luominen dynaamisen luomisen sijaan aina, kun se on mahdollista.

3.5.3 Dynaamisesti luodut taulukot

Joskus tulee tarve luoda dynaamisesti myös taulukoita. Tällöin kannattaa pyrkiä mahdollisuuksien mukaan käyttämään STL:n taulukkoa `vector`, jonka käyttö on paljon mukavampaa ja turvallisempaa kuin C++:n C-kielestä periytyvän perustaulukkotyypin. Näiden `vector`-taulukoiden (tai muiden STL:n tietotyyppien) dynaaminen luominen tapahtuu normaalisti **`new`**'llä ja tuhoaminen **`delete`**llä. Niihin ei liity mitään erityisiä virhemahdollisuuksia. Mikäli perustaulukoita täytyy kuitenkin luoda dynaamisesti, se tehdään operaattorilla **`new`**`[]` ("taulukko-**`new`**") seuraavasti:

```
Lista* lista_p = new Lista; // Yksittäinen olio
Lista* listaTaulukko_p = new Lista[10]; // Taulukollinen olioita
```

Suurin virhelähde dynaamisesti luotujen taulukkojen kanssa on, että ne **täytyy** tuhota operaattorilla **`delete`**`[]` ("taulukko-**`delete`**"):

```
delete[] listaTaulukko_p; listaTaulukko_p = 0;
```

Mikäli taulukon yrittää tuhota tavallisella **`delete`**llä, on ohjelman toiminta määrittelemätön, mutta tuskin toivotun mukainen. Ongelmalliseksi tilanteen tekee se, että itse osoittimen tyypistä kääntäjä ei voi mitenkään nähdä, onko osoittimen päässä taulukko vai yksittäinen olio. Tämän vuoksi kääntäjä ei pysty varoittamaan väärästä **`delete`**stä. Myös päinvastainen kielto pätee, eli yksittäisiä olioita **ei saa** tuhota taulukko-**`delete`**llä, vaikka kääntäjä ei tästä varoitakaan.

3.5.4 Virheiden välttäminen dynaamisessa luomisessa

Dynaamisen muistinhallinnan virheet ovat yleisimpiä virhetyyppejä ohjelmoinnissa. Valmiista ohjelmasta virheen löytäminen on usein todella vaivalloista, koska dynaamisen muistinhallinnan virheet ilmenevät yleensä aivan eri paikassa kuin missä itse virhe on. Niinpä paras tapa virheiden korjaamiseen on niiden syntymisen estäminen jo ohjelmaa suunniteltaessa.

Virheiden syyt ovat usein samoja ohjelmasta toiseen, joten ohjelmointiyhteisön keskuudessa on kehittynyt erilaisia muistisääntöjä ja “kansanperinnettä”, jotka auttavat virheiden ehkäisyssä. Alla esitetään joitain tyypillisimpiä virhetilanteita ja yksinkertaisia sääntöjä, joilla virheiden syntymistä voi pyrkiä välttämään.

Muistivuodot

Ehkä yleisin muistivuotojen syy on, että ohjelmaa suunniteltaessa ei ole suunniteltu sitä, minkä ohjelman osan *vastuulla* dynaamisesti luodun olion tuhoaminen on. Vastuualueiden suunnittelu on muutenkin oliosuunnittelun tärkeimpiä asioita, joten kunnollisella oliosuunnittelulla saadaan eliminoitua myös suuri osa dynaamisen muistinhallinnan ongelmista. Tuhoamisvastuun voi ajatella myös niin, että jokaisen dynaamisesti (ja muutenkin) luodun olion tulisi olla aina jonkin ohjelmanosan tai olion *omistuksessa* ja omistajan vastuulla on myös tuhota dynaamisesti luotu olio.

Dynaamisesti luotujen olioiden hallinnassa pätevät samat säännöt kuin perinteisessä ei-olio-ohjelmoinnissa dynaamisen muistin hallinnassa. Olioajattelu voi kylläkin auttaa paljon sääntöjen noudattamisessa. Tärkeimpiä tällaisia sääntöjä on, että *dynaamisesti luodun olion omistuksen tulisi mielellään säilyä samana olion elinajan.*

Tämä tarkoittaa sitä, että mieluiten saman ohjelman osan tai olion, joka on luonut toisen olion dynaamisesti, pitäisi myös vastata olion tuhoamisesta. Tämän ohjeen noudattaminen helpottaa suunnittelua, koska suunnittelijan ei tarvitse tällöin pitää mielessään, minkä ohjelman osan vastuulla kunkin olion tuhoamisvastuu kulloinkin on.

Jos esimerkiksi olion rakentajassa luodaan dynaamisesti uusi olio, on luonnollista, että se tuhotaan saman olion purkajassa. Vastaavasti jos jostain ohjelmamoduulista löytyy funktio, joka palauttaa paluuarvonaan osoittimen dynaamisesti luotuun olioon, pitäisi samasta ohjelmamoduulista löytyä myös funktio, jonka avulla saatu olio voidaan “palauttaa” moduuliin (toisin sanoen funktio, joka huolehtii olion tuhoamisesta).

Esimerkkinä tästä on vaikkapa STL:n taulukkoluokka `vector`. Mikäli luodaan taulukko `vector<int>`, joka sisältää kokonaislukuja, ei taulukon käyttäjän tarvitse miettiä ollenkaan dynaamista muistinhallintaa, kuten listaus 3.9 seuraavalla sivulla osoittaa. Taulukkoluokan toteutus pitää huolen kaikesta muistinhallinnasta ja piilottaa sen

käyttäjältä. Jos sen sijaan taulukkoon `vector<int*>` talletetaan dynaamisesti luotuja kokonaislukuja kuten listauksessa 3.10 seuraavalla sivulla), nämä kokonaisluvut ovat käyttäjän luomia ja niinpä niiden tuhoamisvastuu on taulukon käyttäjällä, ei itse taulukkoluokalla.

Mikäli olion omistus (vastuu tuhota dynaamisesti luodut oliot) halutaan siirtää ohjelman osalta toiselle esimerkiksi (jäsen)funktiokutsun yhteydessä, tämä tulee dokumentoida näkyvästi funktion rajapintadokumenttiin. Tällöin sekä funktion käyttäjä että toteuttaja ovat ainakin periaatteessa tietoisia siitä, kummalla olion tuhoamisvastuu funktiokutsun jälkeen on. Uudessa C++:n kirjastossa on tällaisia tilanteita varten `auto_ptr`-malli, jota voi käyttää eksplisiittisesti kertomaan, että olion omistus siirretään toiseen paikkaan. Tämän automaattiosoittimen käytöstä kerrotaan lisää aliluvussa 11.7.

Kahteen kertaan tuhoaminen ja käyttö tuhoamisen jälkeen

Toinen yleinen virhe on, että muistivuotojen pelossa liioitellaan tuhoamista ja joko tuhotaan olio kahteen kertaan (yleensä eri osoittimien läpi) tai sitten olio tuhotaan liian aikaisin ja sitä yritetään käyt-

```
 1 void esim()
 2 {
 3   vector<int> taul;
 4
 5   // Luodaan luvut
 6   for (int i=0; i<10; ++i)
 7   {
 8     taul.push_back(i);
 9   }
10
11   // Käytetään lukuja
12   for (int i=0; i<10; ++i)
13   {
14     cout << taul[i] << endl;
15   }
16
17   // Lukujen tuhoaminen vectorin vastuulla, vektori tuhoutuu itsestään
18 }
```

LISTAUS 3.9: Taulukko, joka omistaa sisältämänsä kokonaisluvut

```
1  void esim2()
2  {
3    vector<int*> taul;
4
5    // Luodaan luvut
6    for (int i=0; i<10; ++i)
7    {
8      int* lukup = 0;
9      try
10     {
11       lukup = new int(i); // Luodaan uusi kokonaisluku
12       taul.push_back(lukup);
13     }
14     catch (std::bad_alloc&)
15     {
16       // Muisti lopussa, täytyy siivota jäljet eli jo luodut luvut
17       delete lukup; lukup = 0;
18       for (int j=0; j<i; ++j) { delete taul[j]; taul[j] = 0; }
19       throw;                     // Heitetään virhe edelleen
20     }
21   }
22   // Käytetään lukuja
23   for (int i=0; i<10; ++i) { cout << *(taul[i]) << endl; }
24   // Lopuksi tuhotaan luvut
25   for (int i=0; i<10; ++i) { delete taul[i]; taul[i] = 0; }
26   // Itse taulukko tuhoutuu automaattisesti
27 }
```

LISTAUS 3.10: Taulukko, joka ei omista sisältämiään kokonaislukuja

tää vielä tuhoamisen jälkeen. Tästä aiheutuvat virheet ovat yleensä yhtä vaikeita jäljittää kuin itse muistivuodotkin, ja virheiden vaikutukset saattavat olla paljon mystisempiä.

Samat suunnitteluperiaatteet, jotka estävät muistivuotoja, auttavat yleensä ehkäisemään ennalta nämäkin virheet. Kun ohjelma suunnitellaan niin, että jokaisen dynaamisesti luodun olion omistaa kerrallaan vain yksi ohjelmanosa, ei pelkoa kahteen kertaan tuhoamisesta ole.

Toinen hyvä peukalosääntö on nollata osoittimet aina sen jälkeen, kun niiden päästä tuhotaan dynaamisesti varattu olio:

```
Paivays* p = new Paivays;
// Käytetään päiväystä
delete p; p = 0;
```

Mikäli nyt jo tuhottua oliota yritetään käyttää osoittimen p kautta, on ohjelmalla hyvät mahdollisuudet kaatua tyhjän osoittimen läpi tapahtuvaan muistiviittaukseen — ainakin UNIX-pohjaisissa käyttöjärjestelmissä. Jos p jätettäisiin vanhaan arvoonsa, osoittaisi se suurella todennäköisyydellä siihen osaan muistia, jossa olio on ollut. Tällöin osoittimen läpi tapahtuvat viittaukset saattaisivat jopa toimia jonkin aikaa "zombie-oliota" käyttäen, kunnes kyseinen muistialue otetaan uudelleen käyttöön ja siihen luodaan uusi olio, joka kirjoittaa datansa vanhan datan päälle.

Muistivuodot virhetilanteissa

Vaikka muistivuodot saisikin muuten kuriin, tulee ongelma paljon monimutkaisemmaksi, kun ohjelman pitäisi pyrkiä *toipumaan* virhetilanteista, vaikkapa muistin loppumisesta tai tiedostovirheestä. Yleensä virheen sattuessa ohjelma siirtyy tavalla tai toisella virheenkäsittelykoodiin, ja tätä koodia kirjoitettaessa helposti unohtuu tuhota ennen virhettä dynaamisesti luodut oliot, joita ei enää virheestä toipumisen jälkeen tarvita. Listauksen 3.10 riveillä 14–20 oleva virhekäsittelijä antaa ehkä jonkinlaisen kuvan siitä, mitä kaikkea muistivuotojen estäminen vaatii.[♋]

Virhetilanteista toipumista C++:ssa käsitellään tarkemmin luvussa 11.

[♋] Mainittakoon, että esimerkkiä kirjoitettaessa ohjelmaan tahtoi aina jäädä jokin virhetilanne, jota ei otettu huomioon. Listauksessa oleva kolmas (!) versio tuntuu toimivalta, mutta jos löydät vielä siitäkin muistivuodon, ota yhteyttä.☺

Luku 4

Olioiden rajapinnat

None of us is ever exclusively a student or a teacher. We are always both at the same time, and must always think of ourselves as such. A teacher can get you started on computers, but no one can teach you everything you need to know. At a certain point, you must teach yourself through trial and error, looking over other people's shoulders, futzing around, figuring out what methods work best for you. The same is true in life. A teacher can guide you, but in the end, you have to make your own way.

As you figure things out on your own, you share what you've learned with others. This is the Japanese idea of the sensei. We often see the word translated as "teacher" in English, but literally it means "one who has gone before". In that regard, we are all senseis to somebody.

Zen masters say, "When the student is ready, the teacher will appear." This means two things: When you're ready to learn, all things appear before you as teachers. And when you're ready to teach, the teacher within you appears. The more you recognize the help you've received from those who gave ahead of you, the easier it becomes to give of yourself to those coming up behind.

– Philip Toshio Sudo: Zen Computer [Sudo, 1999]

Kun edellä on käsitelty modulaarisuutta ja olioita, on käsite “rajapinta” tullut esille moneen kertaan. Rajapinta on esitetty sinä olion osana, jonka olion käyttäjä “näkee” ja jota siis pääsee kutsumaan olion ulkopuolelta. Rajapinta on komponentin se osa, jonka avulla ohjelmoijan tulisi pystyä *käyttämään* komponenttia ilman että tietää mitään sen sisäisestä toteutuksesta. Rajapinnasta täytyisi siis löytyä kaikki käyttämiseen tarvittava informaatio. Tämä on jotain enemmän kuin kokoelma jäsenfunktioita — rajapinnasta pitäisi olla myös käyttöohje, joka kertoo miten sitä on tarkoitus käyttää ja miten se käyttäytyy erilaisissa (myös poikkeuksillisissa) tilanteissa.

4.1 Rajapinnan suunnittelu

Rajapinta on muille ohjelmoijille kirjoitettu käyttöohje komponentistamme. Se vastaan kysymykseen “Miten tätä käytetään?”. Rajapinnan taakse kätketty toteutus ei ole rasittamassa tätä käyttöohjetta tai hämäämässä sen lukijaa.

Rajapintojen käyttö ja toteutuksen kätkentä on yksi ehkä tärkeimmistä ohjelmistotuotannon perusperiaatteista, mutta silti sen tärkeyden perustelu uraansa aloittelevalle ohjelmistoammattilaiselle on vaikeaa. Merkityksen tajuaa yleensä itsestäänselvyytenä sen jälkeen, kun on osallistunut tekemään niin isoa ohjelmistoa, ettei sen sisäistä toteutusta pysty kerralla hallitsemaan ja ymmärtämään yksi ihminen. Näissä tilanteissa komponenttijako helpottaa ratkaisevasti, kun yksittäisen ohjelmoijan ei tarvitse jatkuvasti miettiä kokonaisuutta, vaan ainoastaan omaa koodiaan ja niitä rajapintoja ulkopuolelle, joita hän sillä hetkellä tarvitsee.

David Parnas on muotoillut tästä komponenttien suunnittelusta kaksi sääntöä, jotka nykyään yleisesti tunnetaan Parnasin periaatteina (*Parnas’s Principles*) [Budd, 2002]:

- Ohjelmakomponentin suunnittelijan tulee antaa komponentin käyttäjälle kaikki tarvittava tieto, jotta komponenttia pystyy käyttämään tehokkaasti hyväkseen, mutta ei mitään muuta tietoa (komponentista).

- Ohjelmakomponentin suunnittelijalla tulee olla käytössään kaikki tarvittava informaatio komponentille määrättyjen vastui-

den toteuttamiseksi, mutta ei mitään muuta (ylimääräistä) tietoa.

4.1.1 Hyvän rajapinnan tunnusmerkkejä

Rajapintojen suunnittelusta on hyvin vaikeata antaa tarkkoja ohjeita, ja edellä mainittujen Parnasin periaatteiden noudattaminen johtaa melko nopeasti ristiriitatilanteisiin. Kokemus ja tieto lisäävät kuitenkin jokaisella ohjelmoijalla käsitystä siitä, minkä tyyppisiä rajapintoja on helppo, jopa mukava, käyttää, ja mitkä ovat hankalia. Näistä kokemuksista kannattaa ottaa oppia ja pyrkiä itse aina tuottamaan rajapintoja, jotka ovat sillä mukavalla puolella. Ehkäpä osaksi ohjelmoinnin opetusta tulisi ottaa “hyvien ohjelmien ja rajapintojen kirjallisuus”, jossa tutustutaan yleisesti hyviksi todettuihin ohjelmistoihin ja komponenttikirjastoihin.

Seuraava lista ei pyri olemaan täydellinen, vaan on tarkoitettu herättämään ajattelemaan niitä asioita, joita rajapintasuunnittelussa tulisi osata huomioida:

- Noudata hyväksi havaittuja toimintamalleja. Esimerkiksi oheislaitteita kuvaavat rakenteet sisältävät usein seuraavat käyttövaiheet: avaaminen, käyttö ja sulkeminen — tämä on useille ohjelmoijille tuttu ja luonteva toimintaketju. Vastaavasti jos komponentti sisältää paljon erilaisia tietoalkioita tai olioita, niiden luettelointi ja läpikäynti kannattaa toteuttaa kaikissa rajapinnoissa yhtenäisellä tavalla.

- Mieti komponentin käyttö alusta loppuun ulkopuolisen käyttäjän kannalta. Miten komponentti löytyy järjestelmässä? Missä järjestyksessä sen tarjoamia palveluita on tarkoitus käyttää? Ovatko jotkin palvelut käytettävissä vain osan aikaa? Dokumentaatiossa oleva käyttötapaus “normaalikäyttöjärjestyksestä” voi helpottaa rajapinnan ymmärtämistä.

- Dokumentoi komponentin riippuvuudet. Käyttäjälle ei saa tulla yllätyksenä, että komponentti vaatii muita komponentteja toimiakseen, tai että rajapinnan käsittelemillä olioilla pitää olla jokin ominaisuus (esimerkiksi jokainen olio osaa tehdä itsestään kopion).

- Dokumentoi rajapintaan liittyvät elinkaaret ja omistusvastuut. Jos komponentin sisälle annetaan rajapinnan avulla olio, tuhoutuuko se komponentin toimesta (eli omistusvastuu siirtyy) vai täytyykö rajapinnan käyttäjän edelleen huolehtia olion tuhoamisesta?

- Määrittele ja dokumentoi virhetilanteiden käsittely. Pyrkiikö komponentti käsittelemään huomaamansa virhetilanteet itse, vai välittääkö se tiedon niistä edelleen? Millä mekanismilla virheistä ilmoitetaan?

- Mieti rajapinnan operaatioiden nimeäminen tarkkaan. Vältä sanoja, joilla komponentin käyttöalueella voi olla useita merkityksiä. Yleensä ääneen lausuttavissa olevat nimet ovat helpoimmin ymmärrettäviä.

Listalla esiintyy useaan kertaan “dokumentoi”. Tämä korostaa sitä, että äärimmäisen harvoin rajapinnan luettelemat operaatiot (jäsenfunktiot) kertovat riittävästi komponentista. Rajapintaoperaatioiden yhteyteen on liitettävä kuvausta, joka kertoo tarkemmin rajapinnan käytöstä ja käyttäytymisestä kokonaisuutena.

4.1.2 Erilaisia rajapintoja ohjelmoinnissa

Olion rajapinnan määräytyminen ohjelmointikielen tasolla vaihtelee ohjelmointikielestä toiseen. Joissain oliokielissä esimerkiksi jäsenmuuttujat eivät koskaan näy kuin oliolle itselleen, mutta kaikki jäsenfunktiot näkyvät ulospäin. Toisissa kielissä (kuten C++:ssa) taas ohjelmoija päättää näkyvyydestä.

Ohjelmassa saattaa tulla myös tarve siihen, että yhdellä oliolla olisi erilainen rajapinta riippuen siitä, mikä ohjelman osa oliota käyttää. Erilaisia rajapintatarpeita voivat olla ainakin

- luokan “tavallisen” käyttäjän näkemä rajapinta (yleisin, julkinen rajapinta)

- luokan sisäiseen toteutukseen tarvittava rajapinta

- rajapinta, jonka olio näkee toisesta *saman luokan* oliosta

- rajapinta, jonka kantaluokka tarjoaa aliluokalle (aliluku 6.3.1)

- toisiinsa kiinteästi liittyvien luokkien (esim. komponentin ja moduulin) toisilleen tarjoamat rajapinnat (aliluku 8.4)
- rajapinta, jonka läpi ei voi muuttaa olion sisältöä (aliluku 4.3)
- "rajapinta", jonka läpi olioon voi vain viitata, mutta ei käyttää (aliluku 4.4)
- rajapintaluokat tai muut erilliset rajapinnat, jotka luokka lupaa toteuttaa (aliluku 6.9)
- aikariippuva rajapinta, jossa vain osa rajapinnasta on käytettävissä eri ajanhetkillä (esimerkiksi suurin osa tiedostorajapinnan operaatioista on käytettävissä vasta sitten kuin käsiteltävä tiedosto on avattu)
- toteutuksen riippuvuudet määrittelevä rajapinta. Tämä tavallaan käänteinen rajapinta luettelee esimerkiksi ne palvelut, joita toteutus tarvitsee käyttöjärjestelmältä ja kirjastoilta.

Suurin osa oliokielistä tarjoaa mahdollisuuden määrätä vain osan yllämainituista rajapinnoista, ja tavat joilla "erityisrajapintoja" on mahdollista määritellä, vaihtelevat suuresti kielestä toiseen ja ovat enemmän tai vähemmän teennäisiä. Tässä teoksessa käsitellään lähinnä C++-kielen tarjoamia mahdollisuuksia. Jotkin C++:n rajapintaominaisuuksista ovat parempia kuin monissa muissa kielissä, toiset taas selvästi huonompia.

4.1.3 Rajapintadokumentaation tuottaminen ja ylläpito

Rajapintojen dokumentaatio on yksi tärkeimmistä ohjelmoijan työvälineistä, koska moduulien, kirjastojen ja luokkien oikeaa käyttöä ei yleensä pysty päättelemään pelkästä ohjelmakoodista. Dokumentaatiosta tulisi selkeästi käydä ilmi, miten rajapintaa on tarkoitus käyttää ja mitä ehtoja sen käytölle asetetaan (näitä ehtoja käydään tarkemmin läpi aliluvussa 8.1, jossa puhutaan **sopimussuunnittelusta**, *design by contract*).

Koska luokan tai moduulin ulkopuoliset käyttäjät nojautuvat etupäässä rajapinnan dokumentaatioon, on erittäin tärkeää että tämä dokumentaatio päivitetään aina rajapinnan muuttuessa. Rajapintadokumentaatiota tarvitaan usein myös suunnitteluvaiheen lisäksi varsi-

naisessa koodausvaiheessa, kun rajapinnasta täytyy tarkastaa rajapinnan käytön yksityiskohtia kuten parametrien tyyppejä ja järjestystä. Tällöin olisi kätevää että dokumentaatio näyttäisi rajapinnan myös ohjelmointikielen tasolla.

Näistä tarpeista on syntynyt idea kirjoittaa ainakin osa rajapinnan dokumentaatiosta ohjelman kooditiedostojen sisään kommenttien muodossa. Määrämuotoisista kommenteista voidaan sitten sopivalla työkalulla tuottaa automaattisesti ihmiselle helppolukuinen rajapintadokumentaatio. Tämä tietysti helpottaa dokumentaation ajan tasalla pitämistä suuresti, koska dokumentaatio voidaan helposti tuottaa uudelleen rajapinnan muuttuessa. Lisäksi rajapintadokumentaatiosta voidaan tuottaa nettiselaimella käytettävä versio, jolloin sen linkkejä seuraamalla pystyy helposti navigoimaan dokumentaation sisällä.

Java-kielessä työkalu rajapintadokumentaation tuottamiseen on nimeltään Javadoc [Sun Microsystems, 2005], ja se on integroitu osaksi Javan normaalia kehitysympäristöä. Muun muassa Javan omien kirjastojen rajapintadokumentaatiot on yleensä tuotettu Javadocin avulla.

Toinen laajalti käytetty rajapintojen dokumentaatiotyökalu on nimeltään Doxygen [Doxygen, 2005]. Se on ilmainen open source -ohjelma, jonka tukema kielivalikoima on varsin laaja: C++, C, Java, Objective-C, IDL, sekä rajoitetusti PHP, C# ja D (tilanne keväällä 2005). Doxygen tekee myös lähdekoodista rajapintadokumentaation tueksi osittaisia luokkakaavioita, riippuvuusgraafeja, ohjelmalistauksia. Kaikki nämä voidaan tuottaa sekä perinteisinä dokumentteina että selaimella navigoitavassa muodossa.

Vaikka rajapintadokumentaation ylläpitämiseen käytettäisiinkin automaattisia työkaluja, eivät työkalut kuitenkaan vapauta ohjelmoijaa rajapinnan suunnittelusta ja dokumentoinnista. Kaikki aliluvussa 4.1.1 mainitut suunnittelusäännöt pätevät riippumatta siitä, miten rajapintadokumentaatio tuotetaan. Sen sijaan pelkästä dokumentoimattomasta ohjelmakoodista kiireessä työkalulla tuotettu "rajapintadokumentti" saattaa jopa antaa valheellisen kuvan siitä, että rajapinta olisi kunnolla dokumentoitu, vaikka todellisuudessa tuotettu rajapintadokumentaatio toistaakin vain pelkän ohjelmakoodirajapinnan hieman koreammassa muodossa.

4.2 C++: Näkyvyysmääreet

C++:ssa kieli itse ei pakota olion jäsenten näkyvyyttä tiettyyn muottiin. Näkyvyyden säätämistä varten kielessä on avainsanat **public**, **protected** ja **private**. Luokan esittelyssä nämä avainsanat toimivat "otsikkoina", jotka määräävät niiden jälkeen tulevien esittelyjen näkyvyyden. Perinteisesti nämä määreet esiintyvät luokan esittelyssä edellä mainitussa järjestyksessä, mutta tämä on vain tyyliseikka, ei kielen määrämä järjestys. Luokan esittely on tyypillisesti muotoa

```
class Luokannimi
{
public:
  // Tänne tulevat asiat näkyvät luokasta ulos
protected:
  // Tänne tulevat asiat näkyvät vain aliluokille
private:
  // Tänne tulevat asiat eivät näy ulospäin
};
```

Jos luokkaesittelyn alussa ei anneta minkäänlaista näkyvyysmäärettä, on oletuksena C++:ssa **private**, koska se on "tiukin" näkyvyysmääreistä. Vaikka monet oppikirjat (esim. [Stroustrup, 1997]) käyttävätkin tätä hyväkseen ja esittelevät luokan sisäiset jäsenmuuttujat ensimmäisenä ilman mitään näkyvyysmäärettä, ei niiden antamaa esimerkkiä kannata seurata. Luokan esittelyn tarkoituksena on kertoa luokan käyttäjälle, miten luokan olioita käytetään, ja tätä varten käyttäjä tarvitsee luokan julkisen rajapinnan eli **public**-osan. Selkeyden vuoksi on siis syytä kirjoittaa **public**-osa ensimmäisenä, jotta sitä ei tarvitse etsiä luokan sisäisen toteutuksen perästä.

Edellisen koodiesimerkin kommenteissa olevat "selitykset" eri näkyvyysmääreiden merkityksestä ovat vain ylimalkaisia. Alla selostetaan näkyvyysmääreiden **public** ja **private** tarkka merkitys ja yritetään antaa jonkinlainen kuva siitä, miten niitä on tarkoitus käyttää. Määre **protected** liittyy olennaisesti periytymiseen ja käsitellään myöhemmin aliluvussa 6.3.1. Ilman periytymistä **protected** käyttäytyy olennaisilta osin samoin kuin **private**.

4.2.1 `public`

Näkyvyysmääre **`public`** on määreistä yksinkertaisin siinä mielessä, ettei se rajoita jäsenfunktioiden (ja -muuttujien) näkyvyyttä millään lailla, ja niitä voi käyttää missä tahansa ohjelman osassa. Näin luokan **`public`**-osa määrää luokan julkisen rajapinnan, jonka kautta luokan normaali käyttö on tarkoitettu tapahtuvaksi.

Luokan julkisen rajapinnan suunnitteleminen on vaativa tehtävä. Mikäli rajapinnasta jää pois jotain oleellista — esimerkiksi jäsenfunktio jonkin olennaisen asian tekemiseksi —, ei luokan käyttäjällä ole mitään mahdollisuutta korjata puutetta, koska luokan sisäiseen toteutukseen ei pääse käsiksi. Toisaalta julkiseen rajapintaan ei kannata "varmuuden vuoksi" laittaa mitään ylimääräistä. Luokan suunnittelijan kannalta julkinen rajapinta on lupaus luokan tarjoamista palveluista, joten julkiseen rajapintaan laitettujen asioiden pitäisi pysyä muuttumattomina — luokan käyttäjän koodihan riippuu julkisesta rajapinnasta. Näin luokan ylläpidon vuoksi rajapinta pitäisi pyrkiä säilyttämään mahdollisimman yksinkertaisena. Tämä pyrkimys *"minimaaliseen mutta täydelliseen"* rajapintaan onkin usein mainittu hyvän rajapinnan tunnusmerkkinä [Meyers, 1998, Item 18].

Jäsenmuuttujat on syytä pitää visusti poissa julkisesta rajapinnasta useistakin syistä. Ensinnäkin koko olioajattelun peruskivi on sisäisen toteutuksen kätkeminen. Vaikka jäsenmuuttuja olisikin luonteeltaan sellainen, että olion käyttäjän pitäisi päästä käsiksi siihen, ei sitä silti kannata laittaa julkiseen rajapintaan. Tähän on etupäässä kaksi syytä:

- Joskus myöhemmin luokkaa ylläpidettäessä saattaa tulla tarve siirtää kyseinen jäsenmuuttuja jonnekin muualle, esimerkiksi osoittimen päähän olion ulkopuolelle tai kenties korvata koko jäsenmuuttuja jollain toisella rakenteella. Mikäli jäsenmuuttuja on julkisessa rajapinnassa, luokan käyttäjien koodi riippuu siitä eikä sitä voi poistaa.

- Olio itse ei saa mitään tietoa siitä, milloin julkisessa rajapinnassa olevasta jäsenmuuttujasta luetaan tietoa tai milloin siihen sijoitetaan uusi arvo. Tällöin olio ei voi mitenkään reagoida esimerkiksi jäsenmuuttujan arvon vaihdoksiin tai siihen, että jäsenmuuttujan arvoa ylipäätään on kysytty.

Mikäli jotakin jäsenmuuttujaa tunnutaan tarvitsevan luokan julkisessa rajapinnassa, kannattaa ensin miettiä onko tarve todellinen. Yleensä jäsenmuuttujiin liittyvät palvelut tulisi toteuttaa itse luokan jäsenfunktioissa, joten tarve julkiseen jäsenmuuttujaan saattaa olla merkki siitä, että luokan vastuualueeseen kuuluvia palveluita yritetään toteuttaa luokan ulkopuolella. Jos jäsenmuuttujan arvoa todella kuitenkin tarvitaan, kannattaa lisätä julkiseen rajapintaan sopivat "aseta"- ja "anna"-jäsenfunktiot (*setter* ja *getter*), joiden koodissa jäsenmuuttujaan sijoitetaan tai vastaavasti sen arvo luetaan. Näin on tehty esim. `PieniPaivays`-luokassa, josta löytyvät asetus- ja lukufunktiot päivälle, kuukaudelle ja vuodelle (listaus 2.1 sivulla 62).

Jos julkiseen rajapintaan pyrkivä jäsenmuuttuja on olio, eivät asetus- ja lukufunktiot yleensä riitä kattamaan jäsenmuuttujaolion käyttötarvetta. Tällöin on joskus kätevää kirjoittaa "anna"-jäsenfunktio, joka palauttaa *viitteen* jäsenmuuttujaan. Funktion paluuarvon avulla pääsee käsiksi jäsenmuuttujaolioon, mutta jäsenmuuttujaa ei silti tarvitse siirtää **`public`**-puolelle. Listauksessa 4.1 seuraavalla sivulla on esimerkki luokan `Kirja` jäsenfunktiosta `annaPalautusPvm`, jonka kautta kirjan palautuspäivämäärään pääsee käsiksi. Tällaisten funktioiden kautta olion käyttäjä pääsee edelleen muuttamaan jäsenmuuttujan arvoa olion huomaamatta. Tämä voidaan estää palauttamalla jäsenfunktiosta vakioviite, jolloin jäsenmuuttujaoliota ei voi viitteen läpi muuttaa (vakioviitteet käsitellään aliluvussa 4.3.3).

4.2.2 `private`

Näkyvyysmääre **`private`** on C++:n määreistä kaikkein rajoittavin. Jäsenmuuttujiin ja jäsenfunktioihin, jotka on esitelty **`private`**-osassa, pääsee käsiksi vain *saman luokan* jäsenfunktioiden koodissa (ja ystäväfunktioissa ja -luokkissa, joista kerrotaan enemmän aliluvussa 8.4). Koska koko olioajattelun yksi lähtökohdista on ollut olion sisäisen toteutuksen kätkeminen, tulisi luokan jäsenmuuttujien ***aina*** olla kapseloituna luokan **`private`**-osaan.

Luokan jäsenfunktioiden toteutuksessa tulee usein tarve kirjoittaa "apufunktioita", joita kutsutaan useista eri jäsenfunktioista. Nämä on kätevää kirjoittaa luokan **`private`**-puolelle "yksityisiksi" jäsenfunktioiksi. Tällöin luokan omat jäsenfunktiot pääsevät kutsumaan niitä, mutta apufunktiot eivät silti kuulu luokan julkiseen rajapintaan, joten niihin ei pääse käsiksi luokan ulkopuolelta.

```
1  class Kirja
2  {
3  public:
       ⋮
4    PieniPaivays& annaPalautusPvm();
5  private:
       ⋮
6    PieniPaivays palautuspvm_;
7  };
       ⋮
8  PieniPaivays& Kirja::annaPalautusPvm()
9  {
10   return palautuspvm_;
11 }
       ⋮
12 int aikaaJaljella(Kirja& kirja)
13 {
14   return kirja.annaPalautusPvm().paljonkoEdella(tanaan);
15 }
```

LISTAUS 4.1: Jäsenfunktio, joka palauttaa viitteen jäsenmuuttujaan

C++:ssa (ja Javassa) luokan **private**-osaan pääsee käsiksi olion omien jäsenfunktioiden koodi. Lisäksi myös toiset *saman luokan* oliot voivat käsitellä toistensa **private**-osia. Käytännössä tämä tarkoittaa sitä, että jos olion jäsenfunktio saa käyttöönsä toisen saman luokan olion esimerkiksi parametrina, se pääsee käsiksi myös tämän toisen olion **private**-osaan ja siis myös jäsenmuuttujiin. Luokan `PieniPaivays` jäsenfunktio `sijoitaPaivays` on esimerkkinä tästä listauksessa 4.2 seuraavalla sivulla. Siinä päiväyksen jäsenmuuttujien arvot sijoitetaan toisesta samantyyppisestä oliosta.

Pääsy toisen olion **private**-osaan on tietyllä tavalla luontevaa, koska olion sisäisen toteutuksen piilottaminen toiselta saman luokan oliolta on sinänsä turhaa. Joissain oliokielissä kuten Smalltalkissa on kuitenkin otettu vielä tiukempi kanta, eikä toinen saman luokan olio pääse käsiksi toisen olion jäsenmuuttujiin.

```
1 void PieniPaivays::sijoitaPaivays(PieniPaivays& p)
2 {
3   paiva_ = p.paiva_;
4   kuukausi_ = p.kuukausi_;
5   vuosi_ = p.vuosi_;
6 }
```

LISTAUS 4.2: Pääsy toisen saman luokan olion **private**-osaan

4.3 C++: const ja vakio-oliot

Monissa ohjelmointikielissä on jo ammoisista ajoista lähtien ollut mahdollisuus määritellä muuttujien lisäksi vakioita, jotka eroavat muuttujista siinä, että niiden arvoa ei voi muuttaa. C-kieleen tämä mahdollisuus tuli ANSI-standardin mukana 80-luvulla, kun kieleen lisättiin avainsana **const**. C++:ssa **const** on otettu mukaan myös olio-ominaisuuksiin, jossa sen käyttö on osoittautunut erittäin hyödylliseksi.

4.3.1 Perustyyppiset vakiot

C-kielessä perustyyppiset vakiot — lähinnä kokonaisluku- ja liukulukuvakiot — määriteltiin perinteisesti **#define**-esikääntäjäkomennolla. Syynä tähän oli, että vaikka ANSI-standardi toikin kieleen **const**-määreen, sen toiminta oli tehotonta eikä sitä voinut käyttää kaikissa tilanteissa. C++:ssa tilanne on korjattu, eikä **#define**-vakioita ole enää syytä käyttää.

Perustyyppinen vakio määritellään aivan kuten muuttuja, mutta tyypin yhteyteen lisätään määre **const**:

```
1   int const MAX_MJONON_KOKO = 30000;
2   double const PI = 3.14159265;
3   int const RAJA = annaRaja();
4   char mjono[MAX_MJONON_KOKO];
```

const-vakiot käyttäytyvät muuten kuin muuttujat, mutta niihin ei voi sijoittaa. Tämän lisäksi kokonaislukuvakiota voi käyttää myös paikoissa, joissa kääntäjä vaatii käännösaikaisia vakioita, kuten esimerkiksi taulukkojen ko'oissa (katso rivi 4 edellisessä esimerkissä).

C:stä poiketen vakioiden alustusarvon ei tarvitse olla käännösaikainen vaan se voi olla esim. funktion paluuarvo kuten rivillä 3 — tällaista ei-käännösaikaista vakiota ei kylläkään sitten voi käyttää esimerkiksi taulukon kokoa määräämään.

Kielen kannalta on aivan sama, onko sana **const** ennen tyypin nimeä vai sen jälkeen. Aiemmin käytettiin yksinomaan tapaa, jossa **const** tulee ennen tyyppiä (**const int** i), ja tätä tapaa näkee edelleenkin valtaosassa koodia. Tapa laittaa **const**-sana vasta tyypin nimen jälkeen on kuitenkin C++:n kannalta loogisempi, ja sitä näkee käytettävän yhä enemmän uudessa C++-koodissa. (Vastaavat muutkin tyypin määreet kuten osoitin-* ja viite-& tulevat vasta tyypin jälkeen. Tällä on merkitystä kun määreitä on monta peräkkäin.) Tässä teoksessa on siirrytty käyttämään uutta käytäntöä vuoden 2005 painoksesta alkaen.

const-vakiot ovat normaalisti paikallisia siinä käännösyksikössä, jossa ne määritellään. Käytännössä tämä tarkoittaa sitä, että **const**-vakiot voi sijoittaa otsikkotiedostoihin, vaikka sinne ei normaalisti muuttujia laitetakaan.

Usein olio-ohjelmoinnissa vakiot liittyvät kiinteästi jonkin tietyn luokan käyttöön. Tällöin vakiot kannattaa kapseloida luokan sisään luokkavakioiksi, joista kerrotaan tarkemmin aliluvussa 8.2.2.

4.3.2 Vakio-oliot

Olioista voi tehdä "vakio-olioita" aivan samaan tapaan kuin perustyypeistä luodaan vakioita — lisätään olion määrittelyn jälkeen sana **const**:

```
Paivays const joulu(24,12,1999);
```

Olioiden tapauksessa **const**-sanan vaikutus on kuitenkin monimutkaisempi asia. Perustyypeistä puhuttaessa **const**-sanan vaikutus on helppo selittää — muuttujaan ei yksinkertaisesti saa sijoittaa. Olioiden tapauksessa tilanne on paljon hankalampi. Sijoittaminen ei ole välttämättä mielekäs toimenpide kaikille olioille (olioiden sijoittamista käsitellään aliluvussa 7.2). Jos halutaan puhua "vakio-olioista", onkin oleellista se, ettei tällaisen vakio-olion tilaa pystytä muuttamaan. Koska olion tila on kapseloitu olion sisään, voi olion tila muuttua vain jäsenfunktiokutsun seurauksena.

C++:ssa vakio-olioiden käsite on toteutettu jakamalla jäsenfunktiot kahteen ryhmään — niihin, jotka voivat muuttaa olion tilaa ja niihin, jotka eivät. Tämän jälkeen on määrätty, että vakio-olioille saa kutsua vain niitä jäsenfunktioita, jotka eivät voi muuttaa olion tilaa. Saman asian voi myös ajatella niin, että C++:ssa vakio-olioilla on erilainen rajapinta, joka on vain osajoukko luokan koko rajapinnasta.

Jäsenfunktiot, joiden ei haluta muuttavan olion tilaa, merkitään määreellä **const**, joka tulee jäsenfunktion parametrilistan perään sekä luokan esittelyssä että jäsenfunktion määrittelyssä. Listauksessa 4.3 on luokan `Paivays` esittely, jossa päiväystä muuttamattomat jäsenfunktiot on merkitty vakioiksi. Listauksessa on myös esimerkkinä yhden jäsenfunktion määrittely.

Kääntäjä pitää huolen siitä, että **const**-sanan mukanaan tuomaa

```
 1 class Paivays
 2 {
 3 public:
 4   Paivays(unsigned int p, unsigned int k, unsigned int v);
 5   ~Paivays();
 6
 7   void asetaPaiva(unsigned int paiva);
 8   void asetaKk(unsigned int kuukausi);
 9   void asetaVuosi(unsigned int vuosi);
10
11   unsigned int annaPaiva() const;
12   unsigned int annaKk() const;
13   unsigned int annaVuosi() const;
14
15   void etene(int n);
16   int paljonkoEdella(Paivays const& p) const;
17
18 private:
19   unsigned int paiva_;
20   unsigned int kuukausi_;
21   unsigned int vuosi_;
22 };
```

.... määrittely ....

```
 1 unsigned int Paivays::annaPaiva() const
 2 {
 3   return paiva_;
 4 }
```

LISTAUS 4.3: Päiväysluokka **const**-sanoineen

"vakioisuutta" noudatetaan. Jo aiemmin on mainittu, että vakio-olioille voi kutsua vain vakiojäsenfunktioita. Tämän lisäksi kääntäjä pyrkii valvomaan, että vakiojäsenfunktioiden koodissa ei muuteta olion tilaa. Tämän se tekee asettamalla seuraavat rajoitukset vakiojäsenfunktion koodille:

- Vakiojäsenfunktion koodissa ei voi muuttaa jäsenmuuttujien arvoja (toisin sanoen jäsenmuuttujat käyttäytyvät ikään kuin ne olisi määritelty **const**-määreellä).
- Vakiojäsenfunktion koodissa voi kutsua omalle oliolle vain toisia vakiojäsenfunktioita. Tämä rajoitus koskee siis vain jäsenfunktion omaan olioon kohdistuvia kutsuja.
- Vakiojäsenfunktion koodissa osoitin **this** on tyyppiä "osoitin vakio-olioon" (katso seuraava aliluku).

Edellä mainitut rajoitukset eivät kuitenkaan riitä varmistamaan täydellisesti, ettei vakiojäsenfunktiossa olion tila muutu. Osa olion tilaan kuuluvasta tiedostahan saattaa nimittäin olla olion ulkopuolella esim. osoittimien päässä. Tällaiseen dataan eivät kääntäjän tarkastukset ulotu, vaan ohjelmoijan on itsensä pidettävä huoli siitä, ettei vakiojäsenfunktiossa muuteta mitään sellaista, jonka katsotaan kuuluvan olion tilaan.

Joskus harvoin saattaa olla aihetta määritellä jäsenmuuttuja, jota voisi muuttaa myös vakiojäsenfunktioissa. Tällainen on perusteltua vain, jos jäsenmuuttujan muuttaminen ei vaikuta olion "todelliseen" tilaan. Tällaisia jäsenmuuttujia on mahdollista saada aikaan lisäämällä niiden esittelyn eteen avainsana **mutable**.

C++ antaa myös mahdollisuuden siihen, että luokka tarjoaa kaksi samannimistä jäsenfunktiota *samoilla parametreilla*, jos toinen on vakiojäsenfunktio ja toinen ei. Tällaisessa tapauksessa jäsenfunktion kutsuminen toimii niin, että vakio-olioille ja vakio-osoittimien ja -viitteiden läpi kutsutaan vakiojäsenfunktiota, muuten tavallista. Tästä erottelusta on joskus hyötyä, koska vakiojäsenfunktio voi esimerkiksi palauttaa vakio-osoittimen olion dataan, tavallinen jäsenfunktio taas normaalin osoittimen.

4.3.3 Vakioviitteet ja -osoittimet

Varsinaisia vakio-olioita tarvitaan olio-ohjelmoinnissa äärimmäisen harvoin, koska olio-ohjelmoinnin yksi perusajatuksista on, että olioiden tila muuttuu niihin kohdistettujen toimintojen tuloksena. Vakioolion käsite muuttuu kuitenkin erittäin käyttökelpoiseksi, kun otetaan käyttöön vakioviitteet ja -osoittimet.

Vakioviitteellä tarkoitetaan tässä kirjassa viitettä, jonka *läpi* asiat näyttävät vakioilta. Vastaavasti vakio-osoitin on osoitin, jota käytettäessä sen päässä oleva asia vaikuttaa vakiolta. Termiä "vakio-osoitin" ei tule sekoittaa termiin "osoitinvakio", jolla tarkoitetaan osoitinta, joka *itse* on vakio, ts. osoitinta ei voi muuttaa osoittamaan toiseen paikkaan. Huomaa, että kaikki alla esitetyt asiat pätevät niin vakioosoittimille kuin vakioviitteille, vaikka tekstissä mainittaisiinkin vain toinen.

Osoittimista ja viitteistä saadaan vakio-osoittimia ja -viitteitä lisäämällä niiden esittelyyn määre **const**:

```
char const* mjono;
Paivays const& p;
```

Vakio-osoittimia käytettäessä osoittimen päässä oleva olio tai data käyttäytyy *ikään kuin* se olisi vakio — riippumatta siitä, onko olio tai data alunperin määritelty vakioksi. Tämä siis tarkoittaa sitä, että vakio-osoittimen läpi sijoittaminen ja muiden kuin vakiojäsenfunktioiden kutsuminen on mahdotonta.

Vakioviitteiden hyöty tulee siitä, että niiden avulla voidaan oliosta näkyvä rajapinta rajata sellaiseksi, että olion muuttaminen vakioviitteen kautta tulee mahdottomaksi. Jos esimerkiksi funktio ottaa parametrikseen vakioviitteen päiväysolioon, voi funktion käyttäjä luottaa siihen, että funktiolle välitetyn päiväysolion sisältämä päiväys on varmasti sama myös funktiokutsun jälkeen. Vakioviitteiden käyttö on myös tehokas "dokumentointikeino", jolla voi kertoa, ettei tiettyä oliota tai dataa ole tarkoitus muuttaa. Erityisen tehokkaaksi tämän dokumentointikeinon tekee se, että kääntäjä takaa sen noudattamisen. Esimerkiksi seuraava koodi ei mene kääntäjästä läpi:

```
void muutanKuitenkin(Paivays const& pvm)
{
  pvm.asetaPaiva(1); // KÄÄNNÖSVIRHE: asetaPaiva ei vakiojf.
}
```

C++ sisältää automaattiset tyyppimuunnokset ei-vakio-osoittimista vakio-osoittimiksi (ja sama viitteille), mutta ei toiseen suuntaan:

```
1 char* mjono = "Käytä string-luokkaa char*:n sijaan";
2 char const* vakiomjono = mjono; // Ok: ei-vakio ⇒ vakio
3 char* mjono2 = vakiomjono; // KÄÄNNÖSVIRHE: vakio ⇒ ei-vakio
```

Nämä kielen säännöt tarkoittavat käytännössä, että ei-vakio-olioitakin voi käsitellä ikään kuin ne olisivat vakioita, mutta vakio-olioista ei millään saa tavallisia. Tämä on järkevää, kun muistetaan, että vakio-olion rajapinta on vain osajoukko normaaliolion rajapinnasta.

Vakioviitteiden yleisin käyttökohde on epäilemättä funktioiden ja jäsenfunktioiden parametrit. Mikäli funktio ottaa parametrinaan viitteen olioon, jota sen ei ole tarve muuttaa, tulisi parametriviitteen olla ***aina*** vakioviite. Sama pätee tietysti myös osoittimille. Osoittimia käytetään yleisesti myös olioiden jäsenmuuttujina yms. Tällöin kannattaa miettiä, onko osoittimen läpi tarpeen muuttaa oliota. Jos vastaus on ei, kannattaa käyttää vakio-osoitinta.

Edellä on jo mainittu kaksi vakioviitteiden ja -osoittimien tärkeää käyttösyytä: käyttö dokumentointimielessä rajapintaa rajaamaan ja kääntäjän tekemät tarkastukset siitä, että vakiorajapintaa todella noudatetaan. Kolmas syy käyttää vakioviitteitä esimerkiksi funktioiden parametrina on niin yksinkertainen, että se unohtuu helposti: vakio-olion voi laittaa *ainoastaan* vakioviitteen tai vakio-osoittimen päähän.

Jos funktio ottaa parametrinaan tavallisen viitteen olioon, ei funktiolle voi antaa parametrina vakio-oliota, koska tavallisen viitteen kautta tulisi olion muuttaminen mahdolliseksi. Listauksessa 4.4 seuraavalla sivulla on esimerkki tyypillisestä tilanteesta, joka syntyy kun yhdestä funktiosta unohtuu **const**-sana viitteen edestä pois. Funktio saakoHameenKuukaudessa[ϒ] on kirjoitettu oikeaoppisesti niin, että se ottaa parametrinaan vakioviitteen päiväykseen — eihän funktion ole tarkoitus muuttaa parametrina tullutta päiväystä. Funktion toteutuksessa kuitenkin kutsutaan toista funktiota onkoKarkauspaivaa, jonka kirjoittaja ei ole käyttänyt vakioviitettä parametrina, vaikka karkauspäivän testaamisen ei varmaankaan ole tarkoitus muuttaa testattavaa

[ϒ] Funktio testaa, onko annetusta päivästä kuukauden sisällä mahdollisuus saada hamekangasta. (Karkauspäivänä kosimisesta kieltäytyvän täytyy ostaa kosijalle hamekangas. Epäreilua niitä miehiä kohtaan, jotka eivät käytä hametta.)

päivää. Tämä aiheuttaa käännösvirheen, kun vakioviitettä yritetään antaa parametrina funktiolle, jonka ottama viite ei ole vakio.

Toinen tyypillinen vakio-olioihin liittyvä virhe on, että luokkaa suunniteltaessa unohdetaan varustaa olion tilaa muuttamattomat jäsenfunktiot **const**-määreellä. Tällöin käy niin, että vakioviitteiden kautta oliolle ei voi kutsua ainuttakaan jäsenfunktiota! Aiemmin tässä luvussa ollut `PieniPaivays`-luokka (listaus 2.1 sivulla 62) on esimerkki tällaisesta virheellisestä luokasta, jossa luokan suunnittelijan hutilointi estää luokan käyttäjiä hyödyntämästä kielen turvaominaisuuksia. Tällaisten virhetilanteiden varalle C++-kielessä on tyyppimuunnos **const_cast**, josta kerrotaan aliluvussa 7.4.1.

Joskus hyvin harvoin on tarve saada itse osoittimesta vakio sen osoittaman olion sijaan — toisin sanoen halutaan, että osoitinta ei voi muuttaa osoittamaan toiseen paikkaan. Silloin puhutaan *osoitinvakiosta*. C++:n syntaksi seuraa tässä kohtaa tämän kirjan käytäntöä laittaa **const** aina tyypin jälkeen. Osoittimen saa vakioksi lisäämällä **const**-sanan osoitintyypin tähden jälkeen:

```
char* const osoitinvakio = "Loogista, eikö totta";
```

Vastaavasti osoittimen, jota ei saa muuttaa ja jonka osoittamaa dataa ei myöskään saa muuttaa, tyyppi on **char const* const**.

```
1  #include "paivays.hh"
2
3  bool onkoKarkauspaivaa(Paivays* pvm_p);
4
5  bool saakoHameenKuukaudessa(Paivays const& nyt)
6  { // Karkauspäivä on kuukauden sisällä, jos on helmikuu ja
7     // vuoden sisälle osuu karkauspäivä
8     if (nyt.annaKk() != 2)
9     {
10       return false; // Ei helmikuu
11    }
12    else
13    {
14       return onkoKarkauspaivaa(&nyt); // KÄÄNNÖSVIRHE
15    }
16 }
```

LISTAUS 4.4: Esimerkki virheestä, kun **const**-sana unohtuu

4.4 C++: Luokan ennakkoesittely

Huolellisesta ohjelmiston rakenteen suunnittelusta huolimatta tulee vastaan tilanteita, joissa luokkien välinen assosiaatio on kaksisuuntainen, eli kaksi luokkaa tarvitsee tietoa toisistaan vastuualueensa toteuttamisessa. Esimerkiksi kirjastojärjestelmässä lainaustapahtumaa mallintava luokka tarvitsee tiedon kohteena olevasta kirjasta ja kirja voi sisältää tiedon mihin lainaukseen se liittyy.

C++-ohjelmoinnin kannalta tämä tarkoittaa tilannetta, jossa molempien luokkien tulisi tietää toistensa esittely ennen omaa toteutustaan (kuva 4.1). Tällainen tilanne on mahdoton, koska siinä ennen luokan `Laina` esittelyä (`laina.hh`) yritetään lukea sisään luokan `KirjastonKirja` esittely, jonka alussa luetaan luokan `Laina` esittely, jonka alussa… Tällainen syklinen rakenne johtaa väistämättä siihen, että kääntäjä antaa otsikkotiedostoja lukiessaan virheilmoituksen.

Ratkaisu tähän “kumpi esitellään ensin: muna vai kana?” -ongelmaan on esitellä toisesta luokasta vain tieto sen olemassaolosta (ei kokonaista rakennetta). Tällöin luokan koko esittelyä ei C++:ssa tarvitse lukea sisään ja edellä mainittu ongelma poistuu. Pelkän luokan olemassaolon esittely tehdään C++:ssä **ennakkoesittelyllä** (*forward declaration*), jossa kerrotaan luokasta vain sen nimi:

```
class Laina;
```

Koska ennakkoesitellyn luokan sisäinen rakenne (muistinkulutus), rajapintafunktiot ja periytymissuhteet eivät ole tiedossa, C++ sallii käyttää ennakkoesiteltyä luokkaa vain sellaisissa paikoissa, joissa näitä ominaisuuksia ei tarvita. Käytännössä ennakkoesitellystä luokasta

laina.hh

```
// tarvitaan "KirjastonKirja"
#include "kkirja.hh"

class Laina {
// ...
private:
  KirjastonKirja& kohde_;
};
```

kkirja.hh

```
// käytetään luokkaa "Laina"
#include "laina.hh"

class KirjastonKirja {
// ...
private:
  Laina* kenella_;
};
```

VÄÄRIN!

KUVA 4.1: Väärin tehty luokkien keskinäinen esittely (ei toimi)

voidaan tehdä tämän luokan olioihin osoittavia viitteitä ja osoittimia, mutta näiden läpi ei voi tehdä rajapintakutsuja. Samoin luokkaa voi käyttää funktioiden ja jäsenfunktioiden *esittelyissä* parametrina ja paluuarvona.

Luokan `KirjastonKirja` sisältävässä otsikkotiedostossa voidaan nyt vain ennakkoesitellä luokka `Laina`, jolloin sen koko esittelyä ei tarvitse lukea sisään eikä syklisyysongelmaa tule, kuten listauksesta 4.5 nähdään.

Luokan `Laina` otsikko- ja toteutustiedostossa taas voidaan käyttää luokan `KirjastonKirja` esittelyä ilman ongelmia (kuva 4.2 sivulla 115). Kuvaan on myös merkitty numeroin käännöksen eteneminen tiedostoa `kkirja.cc` käännettäessä.

4.4.1 Ennakkoesittely kapseloinnissa

Ennakkoesittely ei ole ainoastaan suunnittelun silmukoiden "oikomiseen" tarkoitettu menetelmä. Sen avulla voi halutessaan jättää kertomatta rajapinnan käyttäjälle yksityiskohtia tietorakenteiden sisällöistä. Jos rajapinta julkistaa luokasta tai **`struct`**-tietorakenteesta vain ennakkoesittelyn, rajapintaa käyttävä ohjelmoija pakotetaan käsittelemään esiteltyä rakennetta vain rajapintafunktioiden avulla. Vertaa oheista moduuliesimerkkiä (listaus 4.6 seuraavalla sivulla) vastaavaan esimerkkiin Modula-3:lla kirjoitettuna (aliluku 1.6.1 sivulla 51).

Ennakkoesittelyllä voidaan myös vähentää **`#include`**-käskyn kautta syntyviä tiedostojen välisiä riippuvuuksia käyttämällä ennakkoesittelyä siellä, missä täydellisen luokkaesittelyn käyttö ei ole aivan välttämätöntä. Tästä ominaisuudesta on esimerkki aliluvussa 9.3.4.

```
1 // kkirja.hh
2 class Laina; // ennakkoesittely
3
4 class KirjastonKirja {
      ⋮
5 private:
6   Laina* kenella_;
7 };
```

LISTAUS 4.5: Esimerkki ennakkoesittelystä

ennakko-paivays.hh

```
1 namespace Paivays {
2   // Ennakkoesittely päiväyksiä kuvaavasta tietorakenteesta:
3   struct PVM;
4   // Palauttaa uuden päiväyksen. HUOM! Tuhottava rutiinilla ''Tuhoa()''
5   PVM* Luo( int paiva, int kuukausi, int vuosi );
6   // Poistaa käytöstä rutiinilla ''Luo()'' käyttöönotetun päiväyksen
7   void Tuhoa( PVM* p );
8 }
```

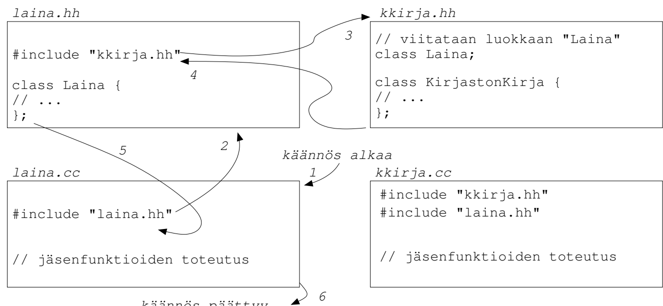
ennakko-paivays.cc

```
1 #include "ennakko-paivays.hh"
2 namespace Paivays {
3   // Päiväysten tietorakenne:
4   struct PVM {
5     inline PVM( int p, int k, int v );
6     int p_, k_, v_ ;
7   };
8   inline PVM::PVM( int p, int k, int v ) : p_(p), k_(k), v_(v) {}
9
10  PVM* Luo( int paiva, int kuukausi, int vuosi )
11  {
12    return new PVM( paiva, kuukausi, vuosi );
13  }
14
15  void Tuhoa( PVM* p )
16  {
17    delete p;
18  }
19 }
```

ennakko-kaytto.cc

```
1 #include "ennakko-paivays.hh"
2
3 void kaytto() {
4   Paivays::PVM* vappu = 0;
5   vappu = Paivays::Luo( 1, 5, 2001 );
      ⋮
6   Paivays::Tuhoa( vappu );
7 }
```

LISTAUS 4.6: Ennakkoesittely kapseloinnissa



KUVA 4.2: Oikein tehty luokkien keskinäinen esittely

Luku 5

Oliosuunnittelu

Koska eräät suunnittelun piirteet eivät ole analyyttisiä tai tieteellisiä, suunnittelun opetus ei yleensä sovi hyvin arvovaltaisiin teknisiin korkeakouluihin, joiden työntekijät tuntevat olonsa kotoisammaksi opettaessaan matematiikkaa ja tieteitä kuin kertoessaan, miten yksilön fysikaalisen maailman ilmiöitä koskevaa arvostelukykyä, estetiikkaa, luovuutta ja herkkyyttä kehitetään. Monet tekniikan professorit toivovat, että suunnittelu olisi tieteellisempää. Itse asiassa tekniikan piirissä on ajoittain liikkeitä, joiden tavoitteena on tehdä suunnittelusta analyyttisempää. Ne epäonnistuvat suunnittelun pehmeämpien osien suhteen. Käytännössä suunnittelu on huomattavasti kriittisempää ja arvostetumpaa toimintaa kuin sen asema tekniikan koulutuksessa antaa ymmärtää.

– Insinöörin maailma [Adams, 1991]

Ohjelmistomäärittely ja -suunnittelu on hyvin laaja-alainen alue, johon on olemassa erilaisia teorioita, menetelmiä ja kansanperinnettä. Kokonaiskuvan alueesta saa esim. teoksesta “Ohjelmistotuotanto” [Haikala ja Märijärvi, 2002]. Kannattaa myös pitää aina mielessä, että ohjelmistojen valmistaminen ei ole ainoastaan tietojenkäsittelyä — ohjelmiston “sieluna” olevan toimintojen filosofian määrittelee aina ihminen ja lähes aina ihmisiä varten [Roszak, 1992]. Tässä luvussa esitellään muutamia oliosuunnitteluun sopivia yleisiä periaatteita ja kuvaustapoja.

5.1 Oliosuunnittelua ohjaavat ominaisuudet

Ohjelmiston jako moduuleihin ja luokkiin on ratkaisu, jota ohjaavat monet tekijät: ongelman ratkaisu, käytetyt työkalut, menetelmät, kokemus, "talon perinteet", reunaehdot, jatkokehityksen suunnitelmat jne. Luokkahuone-esimerkkejä laajemmissa ohjelmistoissa ei koskaan ole olemassa vain yhtä ainoata oikeata tapaa tehdä moduuli- ja luokkajakoa. Seuraavat aliluvut esittelevät niitä asioita, joita oliosuunnittelussa tulisi osata ottaa huomioon [Booch, 1987].

5.1.1 Mitä suunnittelu on?

Kaikessa insinöörityössä suunnittelulla pyritään löytämään ratkaisu johonkin ongelmaan. Ratkaisun "rakennepiirustusten" avulla voidaan valmistaa haluttu tuote. Suunnittelu on reitti ongelman kuvauksen ja määrittelyn sekä lopullisen tuotteen välillä.

Suunnittelun tarkoituksena on saada aikaan järjestelmä, joka

- toteuttaa ongelman (toiminnallisen) kuvauksen
- voidaan toteuttaa käytössä olevilla raaka-aineilla ja resursseilla
- sopii implisiittisiin ja eksplisiittisiin resurssirajoihin [Booch, 1991, s. 20]
- erityisesti ohjelmistojen tapauksessa lopputuloksen tulisi olla varautunut jatkokehitykseen ja ylläpitoon (esimerkiksi laadukkaan dokumentaation avulla).

Varsinkin järjestelmän "piilo-oletusten" kaivaminen esille osaksi suunnitelmaa on yksi vaikeimmista suunnittelutyön osista.

Ohjelmistosuunnittelu on termi, joka voidaan määritellä tarkoittamaan ohjelmiston vaatimusten ja toiminnallisuuden suunnittelua menetelmällä, jossa lopputuloksena saadaan mahdollisimman helposti toteutettava (ohjelmoitava) suunnitelma (esimerkiksi suunnitteludokumentti). Oliosuunnittelussa tämä tarkoittaa sitä, että *jo suunnitteluvaiheessa on otettava huomioon valitun menetelmän (olioiden) ominaisuudet ja pyrittävä tekemään suunnitelma, joka tukee olioiden käyttöä toteutusvaiheessa*.

Ohjelmistotuotteiden tapauksessa lopputulos on äärimmäisen harvoin kerralla valmis (ja heti perään unohdettu) kokonaisuus vaan

tuotteen julkistuksen jälkeen sitä kehitetään eteenpäin lisäämällä ominaisuuksia ja (valitettavan usein) korjaamalla aikaisempiin versioihin jääneitä virheitä.

5.1.2 Abstraktio ja tiedon kätkentä

Oliosuunnittelun tärkeimpiä työkaluja on jo aikaisemmin esitelty moduulien esittelyn yhteydessä (katso luku 1.3.3). Modulaarisuus on ohjelmiston jakoa paremmin hallittaviin kokonaisuuksiin abstrahoinnin ja tiedon kätkennän avulla. Suunnitteluvaiheessa tämä tarkoittaa ohjelmiston jakamista paloihin joko osittavalla tai kokoavalla jaottelulla.

- **Osittava** (*top-down*). Haetaan järjestelmän suurimmat kokonaisuudet, jotka usein ovat toiminnallisia, kuten käyttöliittymä, tietokanta, syöttö ja tulostus, tietoliikenne ja rajapinnat muihin ohjelmiin. Näitä osakokonaisuuksia jaetaan taas vuorostaan pienempiin paloihin, joista lopulta muodostuu moduuleja ja luokkia.

- **Kokoava** (*bottom-up*). Jos suunnittelun alussa tunnetaan parhaiten joidenkin osajärjestelmien toiminta, niin moduulisuunnittelu voidaan aloittaa niistä ja myöhemmin kerätä näitä osia suuremmiksi kokonaisuuksiksi.

Käytännön suunnittelutyö ei tietenkään seuraa pelkästään jompaa kumpaa näistä tavoista vaan on niiden yhdistelmä. Jaottelun yhteydessä tunnistetaan ohjelmiston staattista rakennetta, josta tulee moduulijako ja dynaamisia rakenteita (olioita), jotka kuvataan luokkina. Koska luokka pystyy ilmaisemaan kaikki moduulin tärkeimmät ominaisuudet, useat oliosuunnittelumenetelmät käyttävät suunnitteluvaiheessa vain luokkia.

5.1.3 Osien väliset yhteydet ja lokaalisuusperiaate

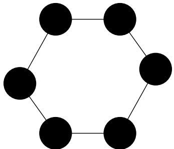
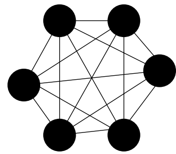
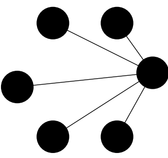
Jotta ohjelmiston moduulien välinen kommunikaatio ei muistuttaisi spagettikoodin aikaisia sotkuja, suunnitteluvaiheessa on pyrittävä pitämään moduulien väliset viittaukset mahdollisimman pieninä. Moduulin A katsotaan viittaavan moduuliin B, kun A tarvitsee jotain

palvelua B:n julkisesta rajapinnasta. Kokoavassa suunnittelussa pyritään keräämään kaikki läheisesti toisiinsa kuuluvat moduulit samaan ylemmän tason kokonaisuuteen (esimerkiksi kaikki ajanlaskuun ja kalenteriin liittyvät moduulit). Tämä vahvasti kytkeytyneiden moduulien paketoiminen uuden pelkistetymmän rajapinnan taakse on esimerkki lokaalisuuden säilyttämisestä suunnittelussa (**lokaalisuusperiaate**).

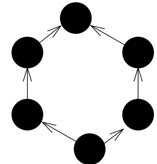
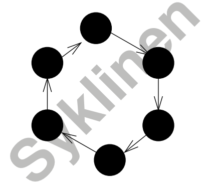
Kuvassa 5.1 on erilaisia riippuvuusvaihtoehtoja moduulien välillä. Jos alijärjestelmässä on n moduulia, niiden välillä on vähintään $n - 1$ riippuvuutta (jokainen osa tietää kokonaisuuden julkisen rajapinnan, joka on yksi osa). Maksimissaan kaikki moduulit tietävät kaikkien muiden olemassaolosta, jolloin riippuvuuksien lukumäärä on $\frac{n(n-1)}{2}$ [Meyer, 1997].

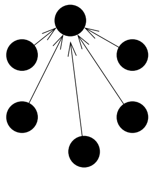
Lokaalisuusperiaate pyrkii minimoimaan ohjelmakomponenttien välisiä yhteyksiä ja näin pitämään kokonaisuuden kompleksisuutta paremmin hallinnassa. Osien välisten riippuvuuksien lukumäärän vähenemisen lisäksi ohjelmiston rakenne yksinkertaistuu, jos riippuvuudet pidetään aina mahdollisuuksien mukaan yksisuuntaisina. Päiväysmoduulin osat eivät tiedä (eivätkä välitä siitä), että niitä käytetään suuremman kokonaisuuden osina. Jos riippuvuudet muodostavat syklisiä silmukkaviittauksia, mutkistuu rakenteiden toteuttaminen: jos A tarvitsee B:tä ja B tarvitsee A:ta, niin kuinka A voidaan toteuttaa ennen kuin on esitelty millainen on B, jota taas ei voida toteuttaa ennen kuin A:n toteutus on valmis jne. (C++:n ratkaisu tähän “muna-kana” -ongelmaan on esitetty aliluvussa 4.4.)

Kuvassa 5.2 seuraavalla sivulla on esimerkkejä erilaisista yksisuuntaisista riippuvuuksista, joista yhdessä riippuvuudet muodostavat silmukan.

KUVA 5.1: Erilaisia yhteysvaihtoehtoja kuuden moduulin välillä



KUVA 5.2: Moduulien välisiä yksisuuntaisia riippuvuuksia

5.1.4 Laatu

Laadukkaisiin suunnitelmiin, moduuleihin ja luokkiin kuuluvia ominaisuuksia on helppo luetella ([Booch, 1987], [Liberty, 1998], [Meyer, 1997]), mutta käytännössä vaikea toteuttaa kaikilta osiltaan:

- **Oikeellisuus.** Ohjelmiston komponenttien (moduuli, luokka tai koko ohjelmisto) tulee toteuttaa kaikki toiminnot suunnitelman määrittelemällä tavalla.

- **Virheitä sietävä.** Komponenttien tulee varautua virhetilanteisiin (väärä syöte, muistin loppuminen jne.) ja osata toimia virhetilanteen havaitessaan etukäteen suunnitellulla tavalla (*robustness*).

- **Selkeys.** Ohjelmiston toiminnallisten yksiköiden rajapintojen dokumentointi ja toteutus on tehtävä yhtenäisellä ja selkeällä menetelmällä, jotta niiden muuttaminen ja uudelleenkäyttö olisi helppoa.

- **Etukäteissuunnittelu.** Moduulien ja luokkien suunnittelussa tulee pyrkiä ottamaan huomioon kohdeprojektin tarpeiden lisäksi yleisempiä näkökohtia, jotta syntyvä komponentti olisi suuremmalla todennäköisyydellä myös käyttökelpoinen osa tulevia projekteja.

- **Tehokkuus.** Komponenttien tulee käyttää tuhlailematta hyväkseen järjestelmän resursseja (prosessointiaika ja muisti).

- **Siirrettävyys.** Komponenttien suunnittelussa ja toteutuksessa tulisi ottaa huomioon, että osia mahdollisesti käytetään tulevaisuudessa niiden kehitysympäristöstä poikkeavassa ympäristössä.

- **Elinkaaren huomioiva.** Laadukkaan ohjelmiston rakenteen ja dokumentaation tulisi tukea jatkokehitystä, jota kaikille "todellisille" ohjelmistoille tullaan jossain vaiheessa tekemään.

"Rational design process and how to fake it"

Tämä *David Parnasin* [Clements ja Parnas, 1986] artikkelin otsikko kuvaa käytännön ohjelmistotyön ristiriitaa usein hyvinkin ylevien suunnitteluperiaatteiden kanssa.

Näitä ristiriitoja on useita. Mikä on oikein toimiva ohjelma? (Asiakkaan ja määrittelijän kanta ei välttämättä ole aina sama.) Mihin kaikkiin mahdollisiin (ja mahdottomalta tuntuviin) virhetilanteisiin ohjelmiston tulisi varautua — ja miten? Kun aikataulut, kiire ja stressi painavat päälle, niin kuka jaksaa ajatella ohjelmiston tulevia ylläpitäjiä ja koodin selkeyttä?

Vaikka käytännön työ aina välillä onkin kaukana akatemian ylevistä päämääristä, niin harva silti väittää, että huolellisesta ohjelmistojen suunnittelusta tulisi luopua kokonaan. Vaikka joissain osissa joudutaankin joustamaan, niin jo pitämällä aina mielessään ylevämpiä päämääriä pystyy varmasti pitämään laatunsa "aloittelijan spagettihäkkyrää" parempana.[♈]

5.2 Oliosuunnittelun aloittaminen

Alku aina hankalaa. Ensimmäisenä täytyy pitää mielessä kaikki kokonaisuuteen liittyvät asiat. Oliosuunnittelun päävaiheet ovat (luettelossa tarkoitetaan komponentilla ohjelmiston moduuleja ja luokkia) [Booch, 1987]

1. komponenttien tunnistaminen

2. komponenttien vastuualueiden määrittely

[♈] Kun joku keksii ideointiin, suunnitteluun, ohjelmointiin ja ihmisten johtamiseen menetelmän, jolla saa ylivoimaista laatua täysin aikataulujen ja budjetin mukaisesti, niin pyydämme nöyrästi kertomaan siitä meille ja muullekin maailmalle.

3. komponenttien välisten suhteiden määrittely (keskinäinen näkyvyys)

4. komponenttien rajapintojen määrittely (esimerkiksi formaalisti matemaattisella notaatiolla, mahdollisimman yksikäsitteisesti ohjelmointikielen rakenteilla tai sanallisella kuvauksella)

5. viimeisenä vaiheena edellä määriteltyjen luokkien ja moduulien sekä niiden muodostaman kokonaisuuden (=ohjelmisto) toteutus.

Luettelo on hyvä esimerkki säännöistä, joissa käytännön suunnittelutyössä ei edetä yksi kohta kerrallaan — “Tämä sääntö mietitään nyt loppuun ennen kuin jatketaan seuraavaan vaiheeseen”. Tärkeä osa suunnittelua on ideointi, jota ei kannata rajoittaa vain yhteen osa-alueeseen kerrallaan. Kun johonkin kohtaan sopiva idea tulee mieleen, niin se kannattaa kirjoittaa muistiin ja luottaa siihen, että myöhemmin suunnittelun viimeistelyssä mahdolliset turhat tai mahdottomat ideat karsiutuvat pois.

5.2.1 Luokan vastuualue

Jokaisesta olio-ohjelmassa olevasta luokasta pitäisi olla olemassa selkeä ja kattava kuvaus. Yhdellä sanalla ilmaistuna on määriteltävä luokan **vastuualue** [Budd, 2002]. Vastuualue määrittelee sen, mitä kyseisen luokan olioiden on tarkoitus mallintaa ohjelmistossa. Vastuualueeseen kuuluvat luokan tarjoamat **palvelut** (jotka ovat käytettävissä luokan julkisen rajapinnan kautta) sekä luokkaan kuuluvan olion toiminnan ymmärtämisen kannalta oleellinen tilatieto eli **attribuutit**. Käytännön luokissa tilatietoon kuuluu usein muutakin kuin yksittäisiä ohjelmointikielen muuttujia. Luokka voi muun ohella omistaa toisia olioita (esim. Päiväys voi sisältyä jonkin toisen luokan tilaan).

Erityisesti kannattaa ottaa huomioon, että suunnitteluvaiheessa on tarkoitus kirjata vain “julkisen” toiminnallisuuden ymmärtämisen kannalta oleellista informaatiota. Esimerkiksi näyttölaitteella olevaa grafiikkapistettä kuvaava luokka sisältää suunnitteluvaiheessa attribuutin “sijainti”, ja tätä tietoa voidaan käsitellä rajapinnan tarjoamien palveluiden avulla. Suunnitteluvaiheessa *ei* ole oleellista määritellä sitä, onko järjestelmässä sijainti-informaation toteutus x- ja y-koordinaatit kokonaislukuina vaiko esimerkiksi polaarikoordinaatiston kul-

ma- ja sädetiedot. Tämä sijaintiattribuutin toteutus on luokan sisäinen asia, ja siihen liittyvät päätökset tulisi tehdä luokan toteutusvaiheessa. (Edelleen käytännön työssä on usein hyödyllistä miettiä myös toteutukseen liittyviä asioita suunnittelun aikana, mutta niiden ei tulisi päästä hallitsemaan ja sotkemaan ylemmän tason suunnittelua.)

5.2.2 Kuinka löytää luokkia?

Helpommin sanottu kuin tehty. Useat oliomenetelmät lähtevät ongelman kuvauksesta. Ohjelmistosuunnittelun käynnistyessä pitäisi olla olemassa vähintään toiminnallinen kuvaus siitä, mitä ollaan tekemässä (sama pätee kaikkeen insinööritoimintaan [Adams, 1991]). Tämän niin sanotun määrittelyvaiheen dokumentista etsitään (vaikkapa alleviivaamalla) substantiivit, joista sitten *kenties* tehdään olioita. Yksi mahdollinen lähtökohta tälle “oliometsästykselle” on **käyttötapaukset** (*use case*), [Jacobson *ja muut*, 1994], joissa on pyritty kuvaamaan yksittäinen ohjelmiston osa **käyttäjäroolin** (*actor*) näkökulmasta. Käyttötapaus pyrkii kuvaamaan lyhyesti, mutta mahdollisimman kattavasti, yksittäiset ohjelmiston toimintaan liittyvät osat. (Näyttävät keskenään ristiriitaisilta vaatimuksilta — ja ovatkin sitä!) Näistä “näytellyistä” ohjelman tärkeimmistä toiminnoista muodostuu määrittely koko ohjelmiston toiminnallisista vaatimuksista. Kuvassa 5.3 seuraavalla sivulla on esimerkki kirjaston tietojärjestelmän yhdestä käyttötapauksesta.

Substantiivien hakumenetelmän hyvänä puolena on, että mallinnettavan ongelman alueelta otetaan sopivan kokoisia kokonaisuuksia ohjelman rakenteeseen. Toisaalta ohjelma tarvitsee luultavasti toimiakseen muitakin osia kuin ne, jotka löytyvät suoraan ongelman kuvauksesta. Emme voi siis olla varmoja, että saamme laadukkaan oliosuunnitelman vain pelkän määrittelyn substantiivien avulla. Esimerkkinä käyttämämme kirjaston toiminnallisuuden kuvauksessa esiintyy varmasti tietoa lainausajoista ja muista vastaavista ajankohdista ja ajanjaksoista. Pelkällä substantiivihaulla emme silti välttämättä keksi, että saatamme tarvita päiväyspalveluita varten oman olion tai moduulin ohjelmistoomme.

Nimi:	Kirjan lainaaminen, versio 1.0 / Jyke
Suorittajat:	Asiakas itsepalvelupäätteellä tai virkailija
Esiehdot:	Lainattavan kirjan ja lainaajan tiedot ovat järjestelmässä, lainauksen suorittaja on kerrottu järjestelmälle.
Kuvaus:	Asiakas tai virkailija syöttää kirjan tiedot järjestelmään joko viivakoodin lukijalla tai näppäinsyötteellä (ISBN-tunniste tai muu yksiselitteinen tieto). Järjestelmä kirjaa lainaustapahtuman asiakkaan lainaustietoihin. Lainaustapahtuman päättyminen kerrotaan lainauksen suorittajalle näyttölaitteen avulla.
Poikkeukset:	Esiehdot eivät ole kunnossa, lainaaja on lainauskiellossa tai kirja on merkitty varatuksi muualle.
Lopputulos:	Kirja on merkitty lainatuksi asiakkaalle määräajaksi.
Muut vaatimukset:	Päivässä pystyttävä käsittelemään 50 000 lainaustapahtumaa, onnistuneen lainauksen on kirjauduttava alle kahdessa sekunnissa, virhetilanteissa käyttäjälle on annettava selvät toimintaohjeet jatkotoimenpiteistä.

KUVA 5.3: Esimerkki käyttötapauksesta

5.2.3 CRC-kortti

CRC-korttimenetelmä on "korttipeli", jossa yhdistetään komponenttien vastuualueiden ja niiden välisten suhteiden miettiminen. Jokaisesta potentiaalisesta komponentista kirjoitetaan kortti, jossa on merkittynä seuraavat asiat: komponentin (olio tai moduuli) nimi (*Component*), kuvaus komponentin vastuualueesta (*Responsibility*) ja lista niistä komponenteista (julkisista rajapinnoista), joita tämä komponentti toimiakseen tarvitsee (*Collaborators*).

Kun komponenttia mietittäessä tulee mieleen uusi mahdollinen komponentti, tehdään siitä heti kortti. Kun kortin jo omaavasta komponentista tulee mieleen sille kuuluva toiminnallisuus, kirjataan tä-

mä vastuu kortille. Jos vastuun toteuttamiseen tarvitaan selvästi toisen komponentin apua, niin tämän yhteistyökumppanin nimi kirjataan myös muistiin. Tätä kuka-mitä -sykliä toistamalla saadaan kirjatuksi hyvin luontevasti eri järjestyksessä tulleita ideoita heti muistiin. Kuvassa 5.4 on esimerkki CRC-kortista. [Beck ja Cunningham, 1989]

CRC-kortteja voidaan hyödyntää myös käyttötapausten kanssa: käyttötapaus käydään lävitse korttien avulla ja tarkastetaan, onko olemassa tarvittavat kortit ja niiden yhteistyökumppanit, joilla käyttötapauksen toiminnallisuus saadaan aikaan (ja jos kaikkia ei ole, niin samalla saadaan mietityksi uusia kortteja) [Wilkinson, 1995]. Vastaavasti tästä korttien "pyörittämisestä" voidaan saada aikaan hyödyllisiä aikaisemmin huomaamatta jääneitä käyttötapauksia. Kun komponentteja on kerätty tarpeeksi, niiden rakennetta ja keskinäisiä suhteita voidaan kuvata tarkemmin seuraavassa aliluvussa esiteltävällä graafisella kuvaustavalla.

Komponentti: Piste	Yhteistyökumppanit:
Vastuut: paikka kuvaruudulla näkyvissä / piilossa paikan muuttaminen väri-informaatio (?)	näytön laiteajuri

KUVA 5.4: Kuva keskeneräisen CRC-korttipelin yhdestä kortista

CRC-korttien haittapuolia ovat huono skaalautuvuus ja ylläpidon vaikeus. Suurissa ohjelmistoissa tulisi “korttipeli”-ideoinnin tasosta riippuen satoja tai jopa tuhansia kortteja, joiden hallinta menee jossain vaiheessa varmasti mahdottomaksi. Käytännön suunnittelun aikana tulee usein myös tarve palata tarkentamaan ja korjaamaan suunnittelun aikaisempaa vaihetta, jolloin CRC-korttien ylläpidosta tulee ongelma. Näiden ongelmien hallitsemiseksi on kehitetty tietokoneella käsiteltäviä oliosuunnitelman graafisia kuvausmenetelmiä, joista tutustumme tarkemmin UML:ään aliluvussa 5.3.

5.2.4 Luokka, attribuutti vai operaatio?

Oliosuunnittelussa tulee (varsinkin aloittelevalle suunnittelijalle) usein vastaan tilanne, jossa ei ole aivan varma siitä, onko käsiteltävä asia luokka vaiko “vain” jonkin luokan attribuuttiominaisuus. Tähänkään suunnittelun osaan ei ole olemassa viisastenkiveä, mutta muutamia ohjenuoria kylläkin.

Toiminnalliset osat, kuten “siirtäminen”, “kasvattaminen” ja “monistaminen” on melko helppo ymmärtää olioiden toimintoina eli niiden luokkien operaatioina. Ominaisuudet kuvaavat jotain jo olemassa olevaa, joten ne sopisivat luokkien attribuuteiksi: “väri”, “sijainti”, “koko”, “ikä” jne. *Kaikki ne osat, joilla on ominaisuuksia ja joita käytetään operaatioiden avulla, ovat olioita (suunnittelun luokkia)*. [Koskimies, 2000]

5.3 Oliosuunnitelman graafinen kuvaus

Unified Modelling Language (UML) [OMG, 2002b] on nykyisin eniten huomiota saanut oliosuunnitelmien graafinen kuvausnotaatio. UML on *Object Management Group*in vuonna 1997 hyväksymä standardi, jonka viimeisin versio (8.6.1999) on 1.3 [OMG, 2002a].

Seuraavien alilukujen on tarkoitus antaa UML:sta sellainen yleiskuvaus, että myöhemmissä luvuissa sitä hyödyntäen piirretyt kaaviot olisivat ymmärrettäviä. Kattava kuvaus UML:sta löytyy teoksista “The Unified Modeling Language Reference Manual” [Rumbaugh *ja muut*, 1999] ja “The Unified Modeling Language User Guide” [Booch *ja muut*, 1999].

On tärkeätä huomata, että UML on *vain kuvaustapa*. Se ei ole menetelmä, joka ottaisi kantaa siihen, miten luokkia ja niiden välisiä suhteita tulisi suunnitella, mikä olisi hyvä rajapinta tai miten ylipäätään ohjelmisto pitäisi jakaa osiin. UML sijoittuu näin ohjelmointikielten ja suunnittelumenetelmien välimaastoon.

5.3.1 UML:n historiaa

UML on kehittynyt useista 90-luvulla syntyneistä olioiden kuvaus- ja suunnittelumenetelmistä yhdistäen niiden ominaisuuksia. UML:n tärkeimmät edeltäjät ovat

- *James Rumbaughin Object Modeling Technique*, OMT [Rumbaugh *ja muut*, 1991]
- *Grady Boochin* käyttämä esitystapa, joka tunnetaan tekijän mukaan *Boochin notaationa* [Booch, 1991]
- *Ivar Jacobsonin* käyttötapaukset [Jacobson *ja muut*, 1994].

UML:n suunnittelijat ovat asettaneet sille muun muassa seuraavia päämääriä [OMG, 2002b]:

- UML tarjoaa valmiiksi määritellyt visuaaliset rakenteet (ulkoasu ja käyttötarkoitus), joilla suunnittelija voi yhtenäistää ohjelmistosuunnitelmien "rakennepiirustuksia".
- UML sisältää laajennus- ja erikoistamismekanismit, joilla perusnotaatiota pystytään soveltamaan myös erityistilanteissa.
- UML on riippumaton ohjelmointikielistä ja ohjelmiston tuotantoprosessista.
- UML pyrkii tukemaan graafisen suunnitelman automaattista (koneellista) käsittelyä.

UML:n avulla tehdyt ohjelmiston kuvaukset jakautuvat kahteen pääosaan: pysyvää (staattista) perusrakennetta kuvaaviin luokkakaavioihin (ohjelmiston luokat, rajapinnat ja oliot sekä niiden väliset suhteet) ja ajoaikaista käyttäytymistä kuvaaviin (dynaamisiin) kaavioihin. Seuraavissa aliluvuissa esitellään UML:n perusnotaatioita esimerkeillä.

5.3.2 Luokat, oliot ja rajapinnat

UML:ssä luokka kuvataan laatikkona, jossa luetellaan luokan ominaisuudet, joista tärkeimmät ovat luokan nimi, attribuutit, julkisen rajapinnan palvelut ja kuvaus luokan vastuualueesta. Kukin näistä on luokkalaatikossa omassa lokerossaan. Yleensä näkyville piirretään ainakin kolme lokeroa (nimi, attribuutit ja palvelut). Jos jokin näistä kolmesta osasta on kuvatun asian kannalta turha, niin lokero jätetään tyhjäksi.

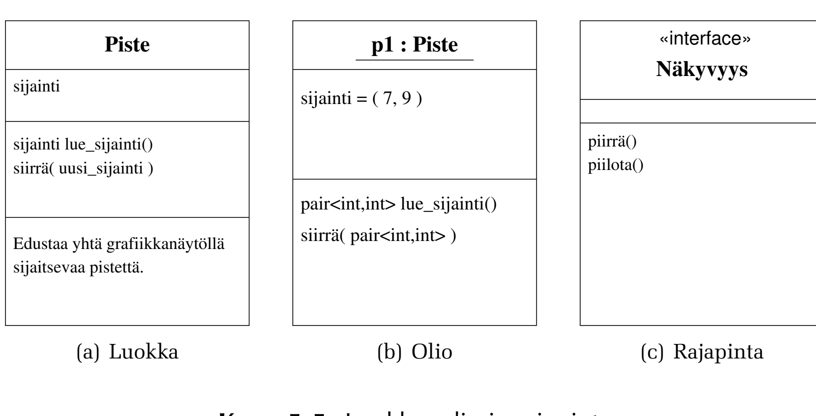
Luokasta tehty olio on kuvaustavaltaan muuten samanlainen, mutta laatikon yläreunassa on alleviivattuna olion nimi (identiteetti) ja tyyppi eli olion luokan nimi. Myös oliolla olevien attribuuttien arvot voidaan merkitä näkyviin.

Usein yksi toiminnallinen rajapinta halutaan kuvata omana yksikkönään. Tällainen julkisen rajapinnan esittelyn sisältävä laatikko merkitään UML:ssä sanalla *interface*. Useampi kuin yksi luokka pystyy lupaamaan, että se toteuttaa jonkin rajapinnan määrittelemän toiminnallisuuden (ja yksi luokka voi toteuttaa useita rajapintoja).

Kuvassa 5.5 seuraavalla sivulla on esimerkki luokasta, oliosta ja rajapinnasta. Luokka kuvaa yksinkertaista grafiikkapistettä, ja siitä on tehty olio, jolle on annettu nimi `p1`. Luokassa on määrätty pisteen vastuulle tieto siitä, missä kohdassa näyttölaitetta se sijaitsee, attribuutilla “sijainti”. Oliosta nähdään, että tämä attribuutti on toteutettu olion sisällä olevilla koordinaattimuuttujilla `x` ja `y`. “Näkyvyys” on rajapinta, joka määrittelee mitä operaatioita näkyvyyttä tukeville olioille voi suorittaa.

UML määrittelee luokan alkioille myös erilaisia näkyvyysmääreitä ja muita niiden kaltaisia ohjelmointikielten rakenteita, kuten jäsenfunktioiden paluuarvoja ja parametreja. Nämä ominaisuudet kertovat jo melko tarkalla tasolla miten luokka halutaan toteuttaa valitulla ohjelmointikielellä. Näitä lisämääreitä käytetään myös hyväksi tietokoneavusteiseen ohjelmakoodin tuottamiseen UML-suunnitelmista.

Ohjelmiston rakentamisen alkuvaiheessa (määrittely, ylimmän tason suunnitelmat ja analyysit) luokkien pitäisi sisältää *ainoastaan* luokan vastuualueen ymmärtämisen kannalta oleelliset asiat (attribuutit, palvelut ja vastuualueen kuvaus). Toteutukseen liittyvien asioiden ei pitäisi olla sotkemassa ylimmän tason suunnitteludokumentteja ja luokkakaavioita.



(a) Luokka (b) Olio (c) Rajapinta

KUVA 5.5: Luokka, olio ja rajapinta

Isoissa ohjelmistoissa ylimmän tason abstrakteja suunnittelukuvauksia usein tarkennetaan lähemmäksi toteutusta ja tällöin myös luokkien ja rajapintojen tarkempi kuvaus on usein käytössä. Kuvassa 5.6 seuraavalla sivulla on esimerkki aikaisemmasta pisteluokasta, jolle on määritelty kaksi jäsenmuuttujaa (`private`-näkyvyydellä) ja julkisen rajapinnan (`public`) jäsenfunktioita. Lisäksi yksi jäsenfunktioista on jätetty ainoastaan luokasta periytettyjen osien käyttöön (`protected`).

Kuvassa on myös taulukko erilaisista UML:n määrittelemistä näkyvyystasoista. Koska näkyvyydet liittyvät läheisesti ohjelmointikieliin, niitä ei kaikissa toteutuskielissä ole suoraan käytettävissä. Esimerkiksi pakkauksen näkyvyystaso on Java:ssa helposti suoraan käytettävissä oleva ominaisuus, mutta C++:ssa se toteutetaan ystävä-ominaisuuden avulla (aliluku 8.4.1 sivulla 259).

UML luokkasuunnittelussa pidetään yleisenä rajanvetona, että kaaviot sisältävät ne asiat, jotka kertovat luokalta vaaditut asiat (vastuualue). Koska toteutus kuitenkin tehdään erikseen ohjelmointikielellä, niin on melko turhaa laittaa suunnittelukaavioon asioita jotka ovat vain ohjelmointia tukevia tai ovat ainoastaan toteutusvaiheeseen liittyviä asioita. Yksi esimerkki tällaisesta UML -kaavion "turhasta" tavarasta (scaffolding code) ovat jäsenmuuttujille halutut aseta ja anna -jäsenfunktiot (aliluku 4.2 sivulla 101). Toteutuksen kannal-

Piste

– x : int
– y : int

+ pair<int,int> lue_sijainti()
+ bool siirrä(int dx, int dy)
# aseta_xy(pair<int,int>)

(a) Toteutusluokka

Näkyvyys	**Merkintä**	**Käytettävissä**
public	+	kaikki ohjelmiston oliot voivat käyttää
protected	#	luokan oliot ja periytettyjen luokkien oliot
private	–	vain samaa luokkaa olevat oliot
package	~	samassa pakkauksessa olevien luokkien oliot

(b) Näkyvyysmääreitä

KUVA 5.6: Tarkennetun suunnitelman luokka ja näkyvyysmääreitä

ta kyseiset jäsenfunktiot voivat olla oleellisia, mutta niiden merkintä luokkakaavioon ei tuo mitään lisäinformaatiota siitä mikä on luokan vastuualue.

5.3.3 Luokkien väliset yhteydet

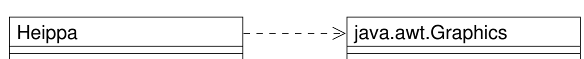
Suunniteltaessa ohjelmiston luokkia on tärkeätä pystyä merkitsemään myös luokkien keskinäisiä yhteyksiä, riippuvuuksia ja niiden kokonaisuuksien toimintoja. Koska ohjelmisto on määrä koota keskenään kommunikoivista olioista, tarvitsemme tavan kuvata näiden olioiden ja niiden luokkien välisiä yhteyksiä. UML:n erilaisia yhteystapoja on koottuna kuvassa 5.7 seuraavalla sivulla.

Yhteys	Tarkenne	Selitys	Symboli
Riippuvuus		Luokka tai moduuli viittaa kohteeseen.	
Assosiaatio		Luokat tai oliot tarvitsevat toisiaan vastuualueensa toteutuksessa.	
	Yksisuuntainen	Vain toinen assosiaation päistä tietää assosiaatiosta (kuka ⇒ ketä -suhde kerrotaan nuolella).	
	Muodostuminen	Olioiden elinkaaret on tiukasti sidottu toisiinsa.	
	Koostuminen	Olioiden elinkaaret liittyvät toisiinsa.	
Periytyminen		Laajennussuhde.	
Toteuttaminen		Luokka toteuttaa erikseen määritellyn rajapinnan.	Rajapintanimi

KUVA 5.7: UML:n yhteystyyppejä

Riippuvuus

Heikoin yhteys on tieto siitä, että luokka A tarvitsee luokan B *olioita* (esimerkiksi rajapintafunktioidensa parametrina). Tämä ilmaistaan katkoviivalla varustetulla nuolella, joka osoittaa luokasta A luokkaan B. Luokan A sanotaan **riippuvan** (*depend*) luokasta B. Kuvassa 5.8 seuraavalla sivulla luokka Heippa vaatii (riippuvuudella ilmaistuna), että luokka `java.awt.Graphics` on sen käytettävissä (vertaa Java-esimerkkiin aliluvussa 1.6.2).



KUVA 5.8: Luokka "Heippa" käyttää Javan grafiikkaolioita

Assosiaatiot

UML:n **assosiaatio** (*association*) on kahden luokan välille piirretty viiva, jonka avulla kerrotaan luokkien liittyvän toisiinsa ohjelmiston rakenteen kannalta. Assosiaatio on riippuvuutta vahvempi ominaisuus (luokat käyttävät toistensa rajapintapalveluita osana oman vastuualueensa toteuttamista). Assosiaatioon merkitään usein tarkennuksia kuvaamaan sen ominaisuuksia. Tärkeimmät tarkennukset ovat assosiaation nimi, lukumääräsuhteet (*multiplicity*) ja luokkien "roolinimet" (*role*). Yleisimmin käytettyjä lukumäärän ilmaisuja UML:n yhteyksissä on esitetty kuvassa 5.9.

Kuvassa 5.10 seuraavalla sivulla on kolme esimerkkiä assosiaatioista. Grafiikkapisteitä ja näyttölaitetta mallintavien luokkien välillä on merkitty tarkennuksena lukumääräsuhteet. Kunkin luokan olioita koskeva lukumäärätieto luetaan assosiaatioviivan toisesta päästä, eli kuvassa on seuraavat lukumääräsuhteet:

- "Näyttölaitteeseen liittyy yksi tai useampi piste" (siis näytöllä on aina vähintään yksi piste).

Merkintä	Selitys
	Jos lukumäärää ei ilmoiteta, se voi olla mikä tahansa
n	Tasan n kappaletta
$*$	Useita (*nolla* tai useampia)
$0..1$	Vaihtoehto (nolla tai yksi kappale)
$m..n$	Lukumäärä ($m \leq$ olioiden lukumäärä $\leq n$)
$1..2, 7, 8..10$	Useita lukumäärävälejä

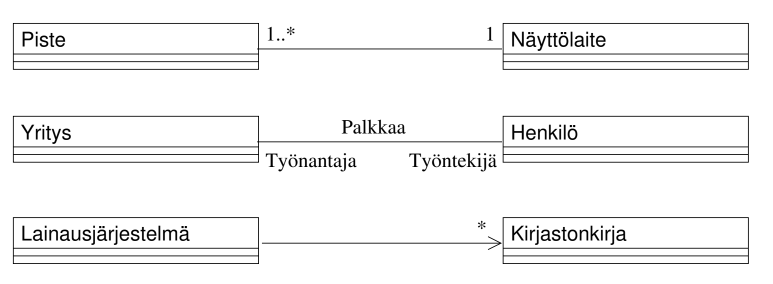
KUVA 5.9: UML-assosiaatioiden lukumäärämerkintöjä

- "Jokaiseen pisteeseen liittyy aina yksi näyttölaite" (siis jokainen pisteolio tietää millä näytöllä se sijaitsee).

Luokkien Yritys ja Henkilö välillä on "palkkaussuhteeksi" nimetty assosiaatio. Tässä tapauksessa Yritys on työnantajan roolissa ja Henkilö työntekijänä.

Usein luokkien välinen yhteys on olemassa vain toiseen suuntaan (lokaalisuusperiaate) eli vain toisen luokan tarvitsee tietää toisen olemassaolosta. Oletuksena UML-assosiaation katsotaan olevan kaksisuuntainen (kumpikin luokka tietää yhteyden olemassaolosta). Jos yhteys on yksisuuntainen, se ilmoitetaan piirtämällä nuoli kohti sitä luokkaa, joka ei tiedä tai välitä assosiaation olemassaolosta (esimerkkikuvassa lainausjärjestelmä käsittelee kirjaston kirjoja, mutta niitä mallintavat oliot eivät tiedä Lainausjärjestelmä-luokan olemassaolosta).

C++: Yksisuuntaiset suhteet pystytään toteuttamaan C++:lla hyvin suoraviivaisesti: toista luokkaa tarvitseva osa ottaa näkyville kohdeluokan esittelyn **`#include`**-komennolla kohdeluokan otsikkotiedostosta ja viittaus käytettäviin olioihin talletetaan osoitin- tai viitejäsenmuuttujaan (listaus 5.1 seuraavalla sivulla). Esimerkissä on valittu talletustietorakenteeksi vektori, koska määrittelyn (kuva 5.10) mukaisesti emme etukäteen tiedä lukumäärän ylärajaa ($*$). Vaihtoehdon (0..1) yleisin toteutus on osoitinjäsenmuuttuja kohdeolioon, joka voi olla arvossa nolla, kun kohdetta ei ole olemassa. Kun lukumäärä on tasan n kappaletta, lukumäärätieto kannattaa kertoa vektorin rakentajan kutsun yhteydessä:



KUVA 5.10: Esimerkki luokkien välisistä assosiaatioista

```
1  #ifndef LAINAUSJARJESTELMA_HH
2  #define LAINAUSJARJESTELMA_HH
3
4  // Lainauksissa käsitellään kirjastonkirjoja
5  #include "kirjastonkirja.hh"
6  class LainausJarjestelma
7  {
        ⋮
8  private:
9    vector<KirjastonKirja*> PikkuKirjastonTietokanta_;
10 };
11 #endif
```

LISTAUS 5.1: Lainausjärjestelmän esittely, `lainausjarjestelma.hh`

```
1  LainausJarjestelma::LainausJarjestelma() :
2    PikkuKirjastonTietokanta_( n )
3  {
        ⋮
```

Kun luokkien välinen assosiaatio on kaksisuuntainen, joudutaan käyttämään aliluvussa 4.4 selostettua ennakkoesittelymekanismia.

Koosteet

Kooste (*aggregate*) on assosiaation erikoistapaus, jonka avulla määritellään luokkien välisen yhteyden lisäksi niiden välille ***omistussuhde***. UML määrittelee kaksi koostetyyppiä: kokoonpanollinen kooste (muodostuminen, *composite aggregate*) ja jaettu kooste (koostuminen, *shared aggregate*).

Koosteen symboli on UML:ssä vinoneliö ("salmiakki"). Salmiakki on assosiaatioviivan päässä kiinni siinä luokassa, jonka osana toisen luokan olio on. Kuvassa 5.11 seuraavalla sivulla on esimerkki koostesuhteista. Umpinaisella mustalla salmiakilla on merkitty, että kirjaston kirja muodostuu yhdestä päiväysoliosta (kokoonpanollinen koostuminen). Kun tämä päiväysolio on liitetty isäntäluokkaansa, vastuu sen elinkaaresta on sidottu isäntäluokkaan. Kun kirjastonkirjaolio tuhoutuu, samalla tuhoutuu myös sen omistama päiväysolio. Tämän

elinkaarisuhteen vuoksi määritellään myös, että yksi olio saa olla muodostumissuhteessa vain yhteen paikkaan.

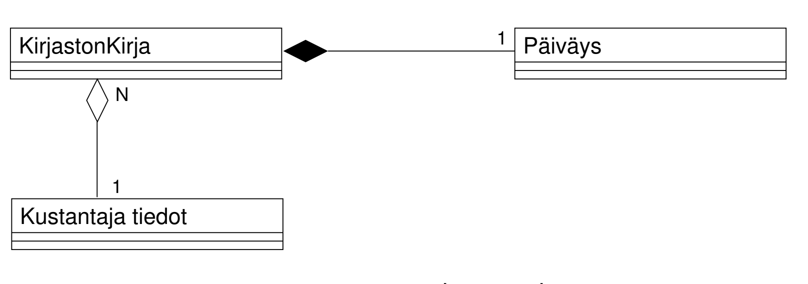
Jaetussa koosteessa (koostuminen) olio voi kuulua useaan isäntäluokkaan ja olion elinkaari ei välttämättä pääty yhtäaikaa emoluokan kanssa. Esimerkkikuvassa jokainen kirjaston kirja koostuu kustantajan tiedot sisältävästä erillisestä oliosta. Sama kustantajaolio voi olla osana useamman kirjan tietoja.

C++: Koosteet toteutetaan C++:ssa jäsenmuuttujien avulla. Jos koostumissuhteen kohde voi muuttua olion elinkaaren aikana (jaettu kooste), toteutuksessa käytetään osoitinjäsenmuuttujaa. Kun koosteoliolle halutaan täsmälleen sama elinkaari kuin emolle, kannattaa C++:ssa käyttää oliojäsenmuuttujaa.

Milloin käyttää koostetta?

UML ei tee ohjelmistosuunnittelijan tehtävää helpoksi määrittelemällä tarkkoja sääntöjä siitä, milloin mitäkin yhteyttä tulisi käyttää (tämä on tietoinen valinta, sillä UML ei ota kantaa käytettyyn suunnittelumenetelmään).

Mielestämme useimmissa tapauksissa ohje yhteyden "lajin" valintaan on selkeä: ***Jos luokka A tarvitsee tiukasti koko elinkaarensa ajan luokkaa B, kyseessä on kooste, muutoin assosiaatio***. Miten sitten määritellään "tiukasti"? Kun koosteen kohteella ei ole enää olemassa yhtäkään "emo-oliota", se on ohjelmiston kannalta turha. Assosiaatiossa olioilla on oma oikeutuksensa olemassaoloon myös yksinään. Kuva 5.12 seuraavalla sivulla yrittää valottaa tilannetta: jos kuvan mukaisessa mallinnuksessa osasto lakkaa olemasta, niin sen



KUVA 5.11: UML:n koostesuhteita

koosteosat eli laitokset lakkaavat myös olemasta (ovat turhia); työntekijäoliot jatkavat kuitenkin olemassaoloaan vaikka niiden työnantaja-assosiaatio poistuisikin.

Koostetyypin (koostuminen tai muodostuminen) valitsemiseen vaikuttaa useimmiten myös kohdeluokan olion vastuualue koko ohjelmistossa. Jos samaa kohdeluokan oliota käyttää yksikin toinen olio tai kohdeoliota ei haluta (esim. operaation raskauden takia) luoda ja tuhota useita kertoja, kyseessä on koostuminen (jaettu kooste).

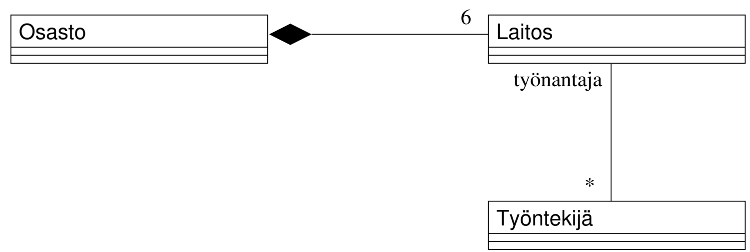
Yleisimmin käytetty yhteys on assosiaatio. Jos halutaan korostaa yhteyden “heikkoutta” tai lyhytkestoisuutta (esimerkiksi kohdeoliota tarvitaan vain parametreina, tai kohteena on tietty järjestelmän ikkunointikirjasto, jonka on pakko olla olemassa toiminnan kannalta), käytetään riippuvuutta.

Periytyminen ja toteuttaminen

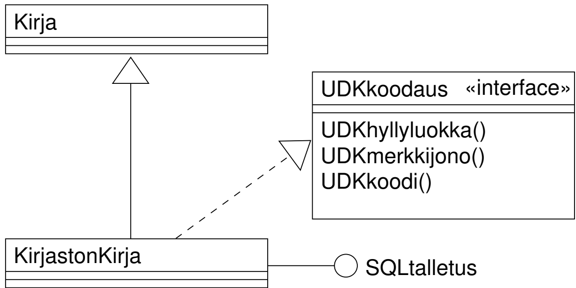
UML:ssä **periytyminen** (*inheritance, generalization*) merkitään nuoliviivalla, joka osoittaa aliluokasta kohti kantaluokkaa. Kuvassa 5.13 seuraavalla sivulla Kirja on kantaluokka, josta on periytetty uusi luokka KirjastonKirja.

.... ***Ekskursio:*** *Periytyminen (lisää luvussa 6)* ....

Periytyminen on olio-ohjelmoinnin ominaisuus, jossa tehdään uusi luokka olemassa olevan mallin (luokan) pohjalta. Periytetty luokka eli **aliluokka** sisältää (perii) kaikki sen mallin eli **kantaluokan** ominaisuudet (attribuutit ja rajapinnan). Aliluok-



KUVA 5.12: Osaston, sen laitosten ja työntekijöiden välisiä yhteyksiä



KUVA 5.13: Kirjasta periytetty luokka, joka toteuttaa kaksi rajapintaa

kaan voidaan lisätä uusia ominaisuuksia, ja kantaluokan ominaisuuksia voidaan tarvittaessa muuttaa.

Koska periytymisellä luodun uuden luokan rajapinta on oletuksena sama kuin kantaluokalla, uuden luokan sanotaan olevan käyttäytymiseltään myös kantaluokan olio (laajennettu versio siitä). Tämä niin sanottu "is-a"-suhde on yksi olio-ohjelmoinnin ja oliosuunnittelun tärkeimmistä ominaisuuksista.

Olio-ohjelmointikielten periytymisominaisuuksista ja niiden vaikutuksista suunnittelupäätöksiin kerrotaan myöhemmin tarkemmin.

. . . .

UML:ssä luokka voi ilmaista kahdella tavalla **toteuttavansa** (*realization, refinement*) tietyn rajapinnan: luokan kylkeen piirretään viivan päässä oleva rajapinnan nimellä varustettu pallo (eräänlainen tikkunekku) tai periytymistä muistuttavalla katkoviivalla. Esimerkkikuvassa KirjastonKirja lupaa toteuttaa UDKkoodaus- ja SQLtalletus-nimisten rajapintojen määrittelemän toiminnallisuuden. Molemmat kuvaustavat tarkoittavat samaa asiaa, ja yleensä kuvaan valitaan siihen parhaiten sopiva tapa (tikkunekku on yleisesti käytössä silloin, kun rajapinta on esitelty kuvan ulkopuolella tai kaukana sitä käyttävästä luokasta).

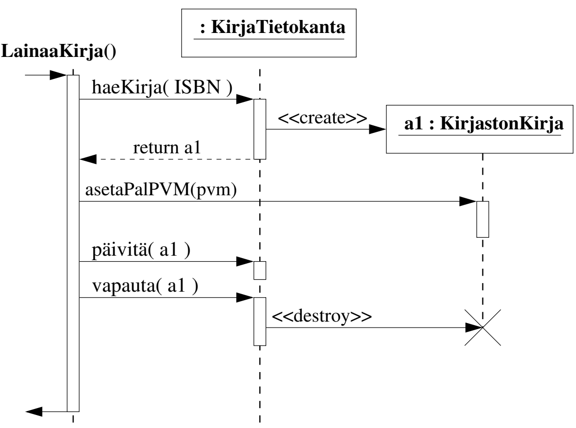
5.3.4 Ajoaikaisen käyttäytymisen kuvaamistapoja

Edellä esitellyillä luokkakaavioilla voidaan kuvata ohjelmiston tarvitsemien luokkien rakenne, niiden väliset yhteydet ja lukumäärät. Nämä ovat staattisia rakennekuvauksia. Kyseessä on kuitenkin vain puolet kattavasta ohjelmiston kuvauksesta. Yhtä tärkeää on kuvata, miten oliot toimivat sisäisesti ja keskenään ohjelman suorituksen aikana. Myös tämän dynaamisen käyttäytymisen kuvaamiseen UML tarjoaa useita kuvaustapoja.

Tapahtumasekvenssit

Lopullisessa ohjelmistossa luokista tehdään olioita, jotka kutsuvat toistensa palveluita jossain järjestyksessä. Näitä ajoaikaisia "suoritusjälkiä" voidaan kuvata UML:lla graafisesti **tapahtumasekvensseillä** (*sequence diagram*). Esimerkki tapahtumasekvenssistä on kuvassa 5.14.

Tapahtumasekvenssikuvan perusideana on piirtää rinnakkain pystyviivoja, jotka kuvaavat aina yhden olion rajapintaa. Kuvassa kulkee aika ylhäältä alas, ja ajan funktiona piirretään rajapintojen välillä tapahtuvia funktiokutsuja. Esimerkissä tarkasteltavalle funktiol-



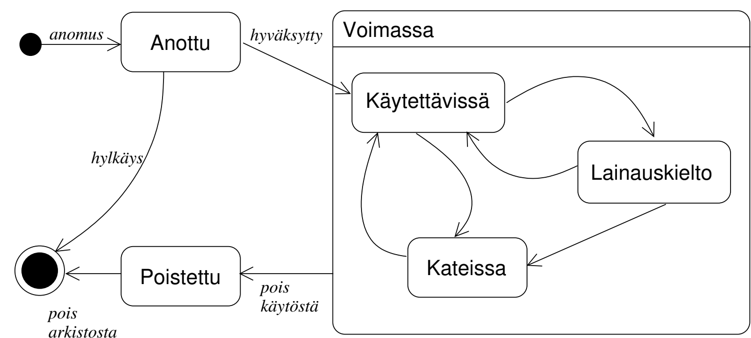
KUVA 5.14: Palautuspäivämäärän asettamisen tapahtumasekvenssi

le (kuvan vasemmassa reunassa oleva pystyviiva) on annettu tehtäväksi päivittää lainattavaan kirjaston kirjaan tieto siitä, koska laina on palautettava. Rutiini pyytää ensin tietokannasta kyseistä kirjaa mallintavan olion, jonka rajapintaan kohdistetaan päivityskutsu (`AsetaPalPVM`). Lopuksi tietokanta päivitetään uuteen tietoon ja operaation suorittamista varten luotu (väliaikainen) olio `a1` tuhotaan. Olion tuhoutuminen eli sen elinkaaren päättyminen ilmaistaan tapahtumasekvenssissä pystyviivan alareunaan piirretyllä ruksilla.

Tilakoneet

Kun olion käyttäytyminen halutaan määritellä tarkemmin kuin sanallisesti ja rajapintafunktioiden kutsujen avulla (tapahtumasekvenssi), voidaan käyttää hyväksi **tilakonetta** (*state machine*).

Tilakoneessa (katso kuva 5.15) on piirretynä toiminnallisia tiloja ja niiden välisiä siirtymiä. Olion (tai yksittäisen rajapintafunktion) kokonaistoiminta kerrotaan kuvaamalla kaikki mahdolliset tilat, joissa se voi olla, ja tapahtumat, joilla siirrytään tilasta toiseen. Olio on aina jossain tilassa, jota nimitetään nykyiseksi tilaksi. Tilasiirtymä tapahtuu, kun siirtymää kuvaavaan nuoleen liittyvä ehto toteutuu. Tilasiirtymässä voidaan kuvata ehdon lisäksi toiminta (esimerkiksi rajapintafunktion kutsu), joka suoritetaan samalla kun olio siirretään siirtymän määräämään uuteen tilaan.



KUVA 5.15: Kirjastokortin tiloja

Tilakoneen mallintama suoritus alkaa aina alkutilasta (musta ympyrä), ja jos nykyiseksi tilaksi tulee tilasiirtymien seurauksena erityinen lopetustila (musta ympyrä valkoisella reunuksella), tilakoneen suoritus päättyy (tilakoneen kuvaaman olion tai funktiokutsun toiminnallisuus päättyy).

Esimerkkikuvassa on mallinnettu kirjastokortin mahdollisia tiloja sen "elinaikana". Kun hakemus kirjastokortista on hyväksytty, kortti aktivoidaan ja suoritus toimii kokoomatilan ("Voimassa") sisällä. Kun kortti poistetaan käytöstä, siirrytään tilaan "Poistettu" riippumatta siitä, missä "Voimassa"-tilan alitilassa tapahtumahetkellä on oltu.

5.3.5 Ohjelmiston rakennekuvaukset

Suurissa ohjelmistoissa luokkiin keskittyvät luokkakaaviot ja tapahtumasekvenssit eivät riitä kuvaamaan ohjelmiston kokonaisuutta tarpeeksi abstraktilla tasolla. Tätä ongelmaa varten UML määrittelee erilaisia arkkitehtuurikuvauksia, joilla on tarkoitus kuvata ohjelmiston rakennetta sen ylimmällä tasolla. (Käyttötapaukset kirjoitetaan usein tämän ylimmän tason terminologiaa ja rakennetta käyttäen.)

Looginen näkymä (*Logical View*) ohjelmistoon on kuva kaikista tärkeimmistä komponenteista ja niiden keskinäisistä suhteista (esimerkiksi käyttöliittymä, ulkopuoliset käyttöliittymäkirjastot, bisneslogiikka, tietokanta ja tietoliikenne). Näiden komponenttien sisällä on luokkakaaviot kunkin komponentin tarkemmasta suunnittelusta.

Jos ohjelmistossa on useita suoritussäikeitä tai prosesseja, **Prosessinäkymä** (*Process View*) kuvaa näiden säikeiden välistä yhteistoimintaa ja keskinäisen kommunikaation protokollaa.

Sijoittelunäkymä (*Deployment View*) näyttää hajautetussa ohjelmistossa sen, missä tietoverkon tietokoneessa (tai vaikkapa UNIXin prosessissa) mikin ohjelmiston osa suoritetaan.

5.4 Saatteeksi suunnitteluun

Suunnittelu on päättymätön matka ja tehtävää suunnitelmaa pystyy "viilaamaan" loputtomiin. Vaikka kuinka haluaisi jatkaa käyttötapausten kirjoittamista aina levyaseman kirjoituspään käyttäjärooliin asti, tulisi käytännön työssä osata lopettaa määrittely ja suunnittelu

joskus. Liiallisuuksiin mennyttä määrittelyä kutsutaan termillä *analysis paralysis* [Liberty, 1998]. Koska sitten on suunniteltu tarpeeksi? Koska perustelimme suunnittelun tärkeyttä sillä, että mutkikas ohjelmisto saadaan jaetuksi yhden ihmisen ymmärtämiin ja toteutettavissa oleviin osiin, niin voimme määritellä lopetusehdoksi tilanteen, jossa kaikki projektin moduulien toteuttajat ovat ymmärtäneet omalta osaltaan vaadittavat vastuut ja osaavat käyttää muiden moduulien julkisia rajapintoja. Kuinka sitten varmistutaan tästä taitaa olla toisen tarinan paikka. . .

Luku 6

Periytyminen

For animals, the entire universe has been neatly divided into things to (a) mate with, (b) eat, (c) run away from, and (d) rocks.

– Equal Rites [Pratchett, 1987]

Yksi ihmismielelle ominainen piirre on pyrkimys kategorisoida — ryhmitellä asioita ja käsitteitä eri luokkiin niistä löytyvien yhtäläisyyksien perusteella. Tätä kategorisointia tekevät jo aivan pienet lapsetkin, ja ihmisten käyttämä kielikin perustuu suurelta osalta sanoihin, jotka eivät tarkoita yksittäistä asiaa vaan koko joukkoa keskenään jollain lailla samankaltaisia asioita.

Kategorisointia käytetään yleisesti myös tieteissä. Biologiassa *Carl von Linné* käytti tätä periaatetta 1700-luvulla jaotellessaan kasveja ja sieniä kategorioihin teoksessaan “Systema Naturae” [Linnaeus, 1748]. On kuitenkin huomattava, että tämä pyrkimys jaotteluun on nimenomaan ihmisen ajattelusta lähtöisin ja että todellinen maailma ei välttämättä aina taivu kovin hyvin tällaisiin malleihin.

Koska jaottelu aliryhmiin erilaisten yhteisten ominaisuuksien perusteella on niin luontevaa ihmisille, se on pyritty ottamaan käyttöön myös ohjelmoinnissa. Olio-ohjelmoinnin painopiste on kahdessa asiassa: olioiden ulkoisessa käyttäytymisessä ja niiden sisäisessä toteutuksessa. Niinpä onkin luonnollista keskittyä kategorisoinnissa näihin aihealueisiin.

Ulkoisen käyttäytymisen jaottelulla saadaan ryhmiteltyä luokkia joukoiksi, jotka joiltain osin käyttäytyvät yhteneväisellä tavalla. Tästä on hyötyä, sillä tällöin sama ohjelmakoodi voi käsitellä kaikkien näiden luokkien olioita, koska olioiden käyttäytyminen on samantapaista.

Sisäisen toteutuksen jaottelussa taas on pyrkimys saada "tislatuksi" eri luokkien yhteisiä osia yhteen paikkaan, jotta samaa ohjelmakoodia ei tarvitsisi kirjoittaa moneen kertaan. Tästä on tietysti selvää hyötyä ylläpidossa ja virheiden korjaamisessa, ja ohjelman kokokin voi pienentyä. Lisäksi koodiin lisätyt uudet luokat voivat ottaa suoraan käyttöönsä olemassa olevien luokkien ominaisuuksia, eli koodia voidaan käyttää uudelleen.

Tälle luokkien jaottelulle ja yhteisistä ominaisuuksista muodostuville "sukulaisuussuhteille" on annettu olio-ohjelmoinnissa nimi **periytyminen** (*inheritance*), ja monet pitävät sitä kapseloinnin ohella olio-ohjelmoinnin tärkeimpänä uutena ominaisuutena perinteisiin ohjelmointikieliin verrattuna. Periytymisen tarkasta merkityksestä ja käyttötavoista keskustellaan ja väitellään olioteoreetikkojen kesken kuitenkin edelleen (matematiikkaa pelkäämättömät voivat tutustua vaikka *Martín Abadin* ja *Luca Cardellin* kirjaan "Theory of Objects" [Abadi ja Cardelli, 1996]).

Periytymisessä on kyse suhteesta, jossa yksi luokka pohjautuu toiseen luokkaan ja perii sen ominaisuudet. Monissa "puhtaissa" oliokielissä jokainen käyttäjän määrittelemä luokka on aina periytetty jostain toisesta luokasta, ainakin kieleen erikseen määritellystä "kaikkien luokkien äidistä" — luokasta `Object` tai vastaavasta. Toisissa kielissä, kuten C++:ssa, tällaista periytymispakkoa ei kuitenkaan ole vaan oletusarvoisesti uusi luokka ei liity mitenkään jo olemassa oleviin luokkiin. Kuvassa 6.1 seuraavalla sivulla on lueteltu tässä teoksessa käytettyjä periytymiseen liittyviä termejä ja selitetty ne lyhyesti.

6.1 Periytyminen, luokkahierarkiat, polymorfismi

Varsinkin eurooppalaiset olio-ohjelmoinnin asiantuntijat korostavat periytymisessä olioiden ulkoisen käyttäytymisen — rajapintojen — kategorista jaottelua [Koskimies, 2000]. Kuvassa 6.2 sivulla 145 on

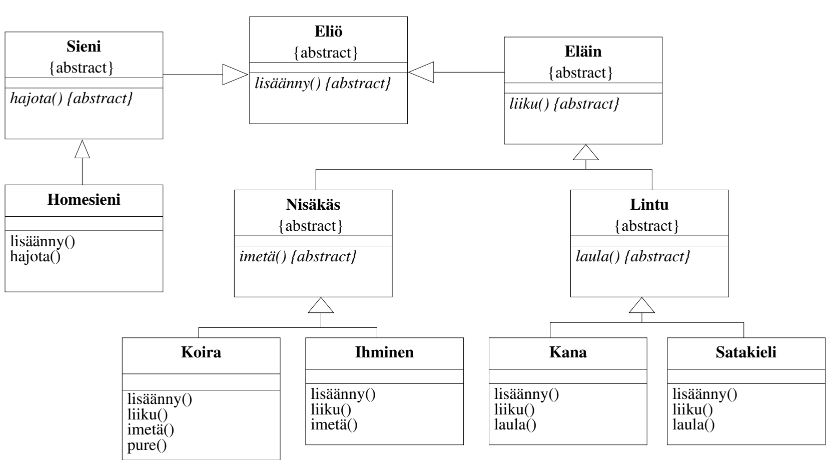
Termi	Selitys
Periytyminen (*inheritance*), (johtaminen)	Uuden luokan muodostuminen olemassa olevan luokan pohjalta niin, että uusi luokka sisältää kaikki toisen luokan ominaisuudet.
Kantaluokka (*base class*), (yliluokka (*superclass, parent class*))	Alkuperäinen luokka, jonka ominaisuudet periytyvät uuteen luokkaan.
Aliluokka (*subclass*), (periytetty/johdettu luokka (*derived class*))	Uusi luokka, joka perii kantaluokan ominaisuudet ja voi tämän lisäksi sisältää uusia ominaisuuksia.
Periytymishierarkia (*inheritance hierarchy*), (luokkahierarkia, (*class hierarchy*))	Kantaluokista ja niiden aliluokista muodostuva puumainen rakenne, jossa puussa alempana olevat luokat perivät ylempänä olevien ominaisuudet. Katso kuva 6.4 sivulla 149.
Esi-isä (*ancestor*)	Periytymishierarkiassa luokan kantaluokka, kantaluokan kantaluokka, tämän kantaluokka jne. ovat luokan esi-isiä.
Jälkeläinen (*descendant*)	Kaikki kantaluokasta periytetyt luokat, niistä periytetyt luokat jne. ovat luokan jälkeläisiä.

KUVA 6.1: Periytymiseen liittyvää terminologiaa

esitetty eliöiden toimintaa mallintavan ohjelman **periytymishierarkiaa** (*inheritance hierarchy*), tosin varsin rajoitetusti.

Tässä esimerkissä vain luokat Homesieni, Koira, Ihminen, Kana ja Satakieli ovat varsinaisia "todellisia eliöitä mallintavia" luokkia, joista ohjelmassa tehdään olioita. Luokat Eliö, Sieni, Eläin, Nisäkäs ja Lintu puolestaan vain kuvaavat todellisten luokkien "yläkäsitteitä". Niitä käytetään ryhmittelemään todellisia luokkia kategorioihin, joihin kuuluvien olioiden rajapinta ja käyttäytyminen ovat jollain tavalla yhteneväisiä.

Tällaisia luokkia, joista ei ole mielekästä tehdä olioita, mutta joita silti tarvitaan kuvaamaan ohjelmassa esiintyvien todellisten luokkien rajapintoja ja niiden suhteita, kutsutaan **abstrakteiksi kantaluokiksi** (*abstract base class*). UML:ssä abstraktit kantaluokat usein merkitään merkinnällä "{abstract}" kuten kuvassa. Samoin merkinnällä



KUVA 6.2: Eliöitä mallintavan ohjelman periytymishierarkiaa

"{abstract}" voidaan vielä merkitä ne rajapinnan jäsenfunktiot, joille abstrakti kantaluokka ei tarjoa toteutusta.

Periytymistä käytetään kuvaamaan luokkien välisiä suhteita. Luokka Eliö kuvaa kaikkia ohjelmassa esiintyviä eliöitä. Sen rajapinta sisältää kaikille eliöille yhteisen rajapinnan, tässä esimerkissä lisääntymisen. Kaikkia esimerkin todellisia olioita voi pyytää lisääntymään, ja jokaisen eliön lisääntymispalvelu näyttää ulospäin täsmälleen samanlaiselta. Sen sijaan lisääntymisen varsinainen toteutus saattaa hyvinkin vaihdella luokasta toiseen — on varsin todennäköistä, että homesieni ja koira lisääntyvät eri tavalla!

Esimerkissä eliöt jaetaan kahteen alikategoriaan, sieniin ja eläimiin. Nämä eroavat toisistaan ulkoisesti siinä, että sieniä voi pyytää hajottamaan eloperäistä materiaalia, eläimiä puolestaan liikkumaan paikasta toiseen. Luokat Sieni ja Eläin on **periytetty** (*derived*) luokasta Eliö. Tällöin niiden rajapintaan kuuluu niiden omien palveluiden lisäksi automaattisesti myös luokan Eliö rajapinta — sienten ja eläinten rajapinta on siis laajennettu eliöiden rajapinnasta. Jälleen kysymys on vain rajapinnasta, ja jokainen todellinen sieniluokka voi toteuttaa hajottamispalvelun haluamallaan tavalla.

Samalla tavoin jaetaan eläimet vielä nisäkkäisiin, joita voi käskeä imettämään, sekä lintuihin, joita voi pyytää laulamaan. Viimein näistä luokista on periytetty todelliset ohjelmassa esiintyvät eläinluokat Koira, Ihminen, Kana ja Satakieli.

Periytymisen hyötynä on, että hierarkia antaa mahdollisuuden puhua esimerkiksi kaikista nisäkkäistä luokkaa Nisäkäs käyttämällä. Jos myöhemmin ohjelmassa tulee esimerkiksi tarve lisätä kaikkien nisäkkäiden rajapintaan uusi palvelu — vaikkapa synnyttäminen —, tämä käy yksinkertaisesti lisäämällä kyseinen operaatio luokan Nisäkäs rajapintaan. Tämän jälkeen täytyy tietysti vielä toteuttaa synnyttäminen kussakin nisäkkäässä lajille ominaisella tavalla.

Vastaavasti jos ohjelmassa on funktio, jonka tehtävänä on siirtää sille parametrina annettuja eläimiä, tämä funktio voi ottaa parametrinaan yksinkertaisesti viitteen Eläin-luokan olioon. Koska sekä Koira, Ihminen, Kana että Satakieli ovat luokkahierarkiassa eläimiä, siirtymisfunktiolle voi tällöin antaa siirrettäväksi *minkä tahansa* eläimen.

Siirtymisfunktion itsensä ei tarvitse tietää yksityiskohtia siitä, minkä eläinlajin edustajaa se on siirtämässä. Sille riittää tieto siitä, että koska kyseessä on eläin, sen julkisessa rajapinnassa on tarvittava palvelu eläimen siirtämiseen. Tällaista tilannetta, jossa funktio hyväksyy eri tyyppisiä parametreja, kutsutaan **polymorfismiksi** (*polymorphism*) eli “monimuotoisuudeksi”. Kääntäjä voi luokkahierarkian avulla lisäksi tarkastaa, että siirtymisfunktiolle annetaan siirrettäväksi vain sellaisia olioita, joita todella voi siirtää — ei esimerkiksi homesieniä. Kielissä, joissa ei ole vahvaa tyypitystä, tarkastetaan sopivan jäsenfunktion löytyminen yleensä vasta ajoaikana. Tällaisissa kielissä luokkahierarkian käyttö rajapintojen luokitteluun on tarpeetonta, koska esimerkiksi siirtymisfunktiolle voisi antaa parametrina minkä tahansa olion, ja jos kyseinen olio ei osaa siirtyä, annetaan ajoaikainen virheilmoitus. Esimerkiksi Smalltalk kuuluu tällaisiin oliokieliin.

Luokkahierarkioiden etuna on vielä se, että jokaisessa todellisessa luokassa rajapintafunktio voidaan tarvittaessa toteuttaa eri tavalla. Edellä mainittu siirtymisfunktio pystyy siirtämään mitä tahansa eläimiä kutsumalla näiden “liiku”-palvelua. Vaikka siirtymisfunktio ei välitäkään siitä, minkä lajin eläimestä on kysymys, voi jokainen eläin silti toteuttaa liikkumisen omalla tavallaan — linnut lentämällä, koira jolkottamalla jne. Näin sama siirtymisfunktiossa oleva “liiku”-palvelun kutsu voi ajoaikana aiheuttaa palvelupyynnön vastaanottajaolion luokasta riippuen erilaisen toiminnon. Tätä kutsutaan

puolestaan **dynaamiseksi sitomiseksi** (aliluku 6.5.2).

Periytymiseen ja hierarkioihin perustuvassa kategorisoinnissa korostuu olion tarjoaman palvelun ja sen toteutuksen ero. Kaikki Eläinluokasta periytetyt luokat sisältävät palvelun “liiku” ja lupaavat, että sitä kutsumalla olio liikkuu paikasta toiseen. Rajapinta ei kuitenkaan lupaa mitään siitä, *millä tavalla* liikkuminen tapahtuu. Tämän voi ajatella myös niin, että hierarkiassa alempana olevat luokat **erikoistavat** (*specialize*) ylempänä määrättyjen palveluiden toimintaa. Erikoistaminen voi olla myös useampivaiheista, esimerkiksi luokassa Lintu saatettaisiin erikoistaa liikkumista määräämällä, että se tapahtuu lentäen. Siitä huolimatta kanat ja satakielet voisivat vielä erikoistaa tätä lisää toteuttamalla lentämisen eri tavalla.

6.2 Periytyminen ja uudelleenkäyttö

Luokkasuunnittelun edetessä tulee vastaan tilanteita, joissa huomataan osan luokkaa (attribuuttien tai toiminnallisuuden) olevan samanlainen useassa luokassa. Esimerkiksi jos mietimme muita grafiikkaluokkia kuin aikaisemmin esimerkkinä ollut Pistettä, niin keksimme ehkä ympyrän, viivan ja valokuvan. Näillä kaikilla on yhteisenä ominaisuutena tieto näkyvyydestä (onko mallinnettu grafiikkaolio piirrettynä näyttölaitteelle), joka ilmaistaan luokan attribuuttina ja sen käyttöön liittyvinä rajapintafunktioina. Attribuutin ilmaisema tilatieto ja siihen liittyvä rajapinta on kaikilla hierarkian luokilla sama.

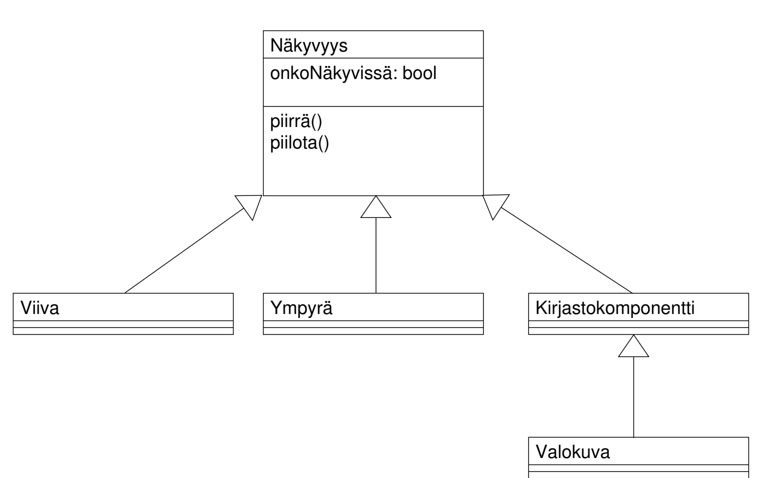
Ohjelmiston ylläpidettävyys paranee, jos kaikille luokille yhteiset attribuutit ja rajapintafunktiot toteutetaan vain kerran ja sama toteutus on kaikkien luokkien käytössä. Oliosuunnittelussa tämä **yleistäminen** (*generalization*) saadaan aikaiseksi määrittelemällä yhteiset osat yhteen luokkaan, josta muut ominaisuutta tarvitsevat luokat *perivät toteutuksen* osaksi itseään. Yhteinen toiminnallisuus tavallaan nostetaan periytymishierarkiassa ylemmälle tasolle yhteiseen kantaluokkaan.

Kuvassa 6.3 seuraavalla sivulla on määritelty kantaluokka Näkyvyys, jonka vastuulle on merkitty tietämys siitä, onko grafiikkaolio näkyvillä näyttölaitteella (attribuutti), ja siihen liittyvät rajapintafunktiot. Kantaluokasta periytetyt luokat perivät *kaikki* kantaluokan ominaisuudet osaksi itseään. Tämä tarkoittaa sitä, että kun Ympy-

rä-luokasta tehdään olio, se sisältää sekä Näkyvyys-luokan että ympyrän ominaisuudet. Tarkasteltaessa periytymistä periytetyn luokan kannalta se on erikoistettu versio kantaluokasta.

Luokkien periyttämistä voidaan jatkaa periytymishierarkiassa eteenpäin. Jos Kirjastokomponentilla on hyvin samanlaiset ominaisuudet kuin näytöllä esitettävällä valokuvalla, voimme periyttää luokan Valokuva Kirjastokomponentista. Valokuva-luokalla on nyt kaikki samat ominaisuudet kuin sen molemmilla kantaluokilla — myös näkyvyyteen liittyvät.

Periytyminen liittyy läheisesti ohjelmakoodin uudelleenkäyttöön. Aikaisemmin olleesta luokasta (mahdollisesti ostetussa komponentissa tai aikaisemmassa projektissa) saadaan käytetyksi kaikki halutut ominaisuudet, ja johdetussa uudessa versiossa tarvitsee ainoastaan lisätä ja muuttaa tarvittavat uudet osuudet. Tämä vahva ominaisuus sisältää myös riskin: pitkät (ja huonosti dokumentoidut) periytymishierarkiat ovat usein perinteistä vaikealukuisempaa ohjelmakoodia, koska hierarkian eri osat (luokat) voivat sijaita eri puolilla ohjelmakoodia (esimerkiksi eri tiedostoissa).

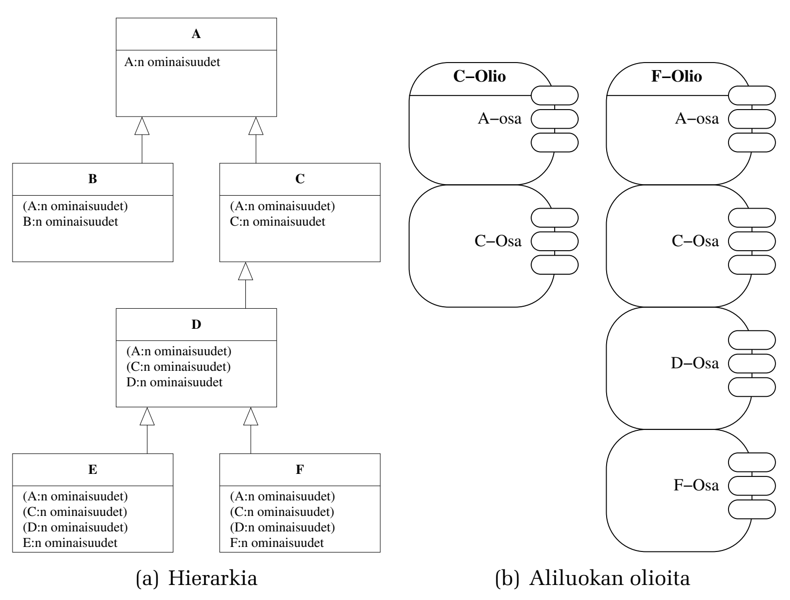


KUVA 6.3: Näkyvyyttä kuvaava kantaluokka ja periytettyjä luokkia

6.3 C++: Periytymisen perusteet

C++:ssa periytyminen tarkoittaa, että kielessä on mahdollista luoda uusi luokka jo olemassa olevaan luokkaan perustuen niin, että uuden luokan oliot perivät automaattisesti kaikki toisen luokan ominaisuudet, niin rajapinnan kuin sisäisen toteutuksenkin. Kuva 6.4 näyttää, mistä periytymisessä yksinkertaistettuna on kyse. Jokainen aliluokka perii kaikki esi-isiensä ominaisuudet ja voi lisäksi laajentaa ja rajoitetusti muokata niitä. Kuvaan on piirretty myös aliluokan olioita, joista näkyy olioiden "kerrosrakenne" — oliot ovat ikään kuin kantaluokan olioita, joihin on liimattu kiinni aliluokkien vaatimat lisäosat.

C++:ssä on myös mahdollista periyttää luokka useammasta kuin yhdestä luokasta. Tämä **moniperiytyminen** (*multiple inheritance*) on varsin kiistelty ominaisuus, ja sen käyttöä ei yleensä suositella kuin tiettyihin tarkoituksiin. Moniperiytymistä käsitellään jonkin verran



(a) Hierarkia (b) Aliluokan olioita

KUVA 6.4: Periytymishierarkia ja oliot [Koskimies, 2000]

aliluvussa 6.7. Aliluvussa 6.9.2 käsitellään myös yhtä moniperiytymisen käyttötapaa — rajapintaluokkia.

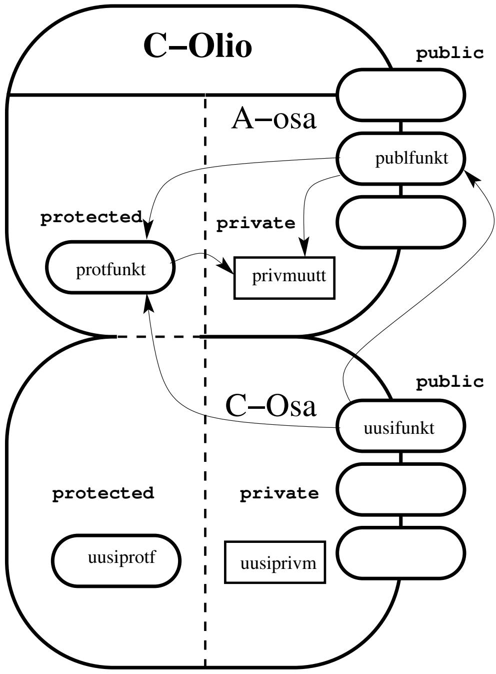
Periytymisen syntaksi on yksinkertainen: aliluokkaa esiteltäessä merkitään luokan nimen jälkeen kaksoispiste ja sen jälkeen periytymistyyppi (lähes aina **public**) ja kantaluokan nimi (moniperiytymisen yhteydessä näitä periytymistyyppi-kantaluokka-pareja on useita pilkulla toisistaan erotettuina). Listauksessa 6.1 on esimerkkinä kuvan 6.4 luokkahierarkiaa C++:lla toteutettuna. C++:ssa on mahdollista public-periytymisen lisäksi myös private- ja protected-periytyminen, mutta niiden käyttö on varsin harvoin olio-ohjelmoinnissa tarpeellista, eikä niitä käsitellä tässä teoksessa.

6.3.1 Periytyminen ja näkyvyys

Periytymisen yhteydessä aliluokka perii kantaluokasta niin ulospäin näkyvän rajapinnan kuin sisäisen toteutuksenkin. Perittyjen osien näkyvyys aliluokan oliossa on kuitenkin osin erilainen kuin kantaluokassa. Kuva 6.5 seuraavalla sivulla kuvaa aliluokan pääsyä kantaluokan eri osiin. Alla on luettelo, josta käy ilmi eri näkyvyysmääreiden vaikutus periytymiseen.

```
1  class A
2  {
3  // Luokan A ominaisuudet
4  };
5  class B : public A
6  {
7  // Luokan B A:han lisäämät ominaisuudet
8  };
9  class C : public A
10 {
11 // Luokan C A:han lisäämät ominaisuudet
12 };
13 class D : public C
14 {
15 // Luokan D C:hen lisäämät ominaisuudet
16 };
       ⋮
```

LISTAUS 6.1: Periytymisen syntaksi C++:lla



KUVA 6.5: Periytyminen ja näkyvyys

- **public:** Kantaluokan public-osat — sen ulkoinen rajapinta — ovat myös aliluokassa public-puolella. Näin aliluokan ulkoisessa rajapinnassa on kaikki se, mitä kantaluokassakin, sekä lisäksi aliluokan määrittelemät uudet rajapintafunktiot.

- **private:** Vaikka kantaluokan private-osa periytyykin aliluokkaan, ei niihin pääse käsiksi aliluokan jäsenfunktioista. Tämä johtuu siitä näkökannasta, että private-osat ovat kantaluokan sisäisenä toteutuksena kantaluokan oma asia, eikä aliluokan tarvitse päästä niihin käsiksi — eihän private-osa muutenkaan näy luokasta ulos. Aliluokan jäsenfunktiot voivat tietysti välillisesti käyttää kantaluokan private-osia kutsumalla kantaluokan rajapintafunktioita.

- **protected:** Tähän saakka protected-määreestä ei ole mainittu juuri mitään muuta, kuin että se on hyvin samantapainen kuin private. Protected-määreen käyttö liittyykin nimenomaan periytymiseen. Kantaluokan protected-osa periytyy aliluokan protected-osaksi. Näin protected-osa on kuin private-osa siinä mielessä, ettei se kuulu luokan ulkoiseen rajapintaan, mutta se on kuitenkin aliluokkien koodin käytettävissä. Protected-osan tehtävä onkin laajentaa kantaluokan rajapintaa aliluokan suuntaan tarjoamalla tälle erityisrajapinnan. Tällä tavoin kantaluokka voi tarjota aliluokille luokan laajentamiseen tarvittavia apufunktioita, joita ei kuitenkaan haluta yleiseen käyttöön. Kuten public-osaankin, protected-osaan tulee kirjoittaa vain jäsenfunktioita, vaikkei C++:n syntaksi tätä rajoitakaan.

Edellä olevista näkyvyyssäännöistä aloittelijaa ihmetyttää usein kantaluokan private-osan "eristäminen" aliluokalta. Kapseloinnin kannalta tämä C++:n (ja Javan) ratkaisu on kuitenkin oikea, koska aliluokkaa ei pitäisi toteuttaa niin, että se riippuu kantaluokan sisäisestä toteutuksesta. Mikäli aliluokan täytyy saada suorittaa sellaisia kantaluokan operaatioita, joita ei löydy kantaluokan julkisesta rajapinnasta, protected-määre tarjoaa käytännöllisen ratkaisun ongelmaan.

Jotkut oppikirjat [Lippman ja Lajoie, 1997] ovat ottaneet sen kannan, että koska kantaluokan suunnittelija harvoin tietää tarkoin, millaisia luokkia kantaluokasta tullaan periyttämään, pannaan varmuuden vuoksi koko kantaluokan sisäinen toteutus protected-puolelle — jäsenmuuttujineen kaikkineen. Tällöin kantaluokan ja aliluokan välinen kapselointi kuitenkin murtuu ja aliluokka pääsee riippumaan kantaluokan sisäisestä toteutuksesta. Paljon parempi ratkaisu on kirjoittaa protected-puolelle sopivat apujäsenfunktiot, joiden avulla aliluokka pääsee sopivasti käsiksi kantaluokan sisimpään.

6.3.2 Periytyminen ja rakentajat

Kuten kaikki oliot, myös aliluokan oliot täytyy alustaa kun ne luodaan. Aliluvussa 3.4 käsiteltiin olion alustamista rakentajajäsenfunktion avulla. Periytyminen tuo kuitenkin omat lisänsä olioiden alustamiseen. Jokainen aliluokan olio koostuu kantaluokkaosasta tai -osista sekä aliluokan lisäämistä laajennuksista. Selvästi aliluokan täytyy määritellä oma rakentajansa, jotta aliluokan uudet osat saadaan alus-

tetuksi, mutta miten pitäisi toimia kantaluokan jäsenmuuttujien alustamisen kanssa? Aliluokan jäsenfunktiothan — rakentaja mukaanlukien — eivät pääse kantaluokan private-osiin käsiksi.

Ratkaisu alustusongelmaan löytyy jälleen luokkien vastuualueista. *Aliluokan vastuulla* on aliluokan mukanaan tuomien uusien jäsenmuuttujien ja muiden tietorakenteiden alustaminen. Tätä tarkoitusta varten aliluokkaan kirjoitetaan oma rakentaja tai rakentajat. *Kantaluokan vastuulla* on pitää huoli siitä, että aliluokan olion kantaluokkaosa tulee alustetuksi oikein, aivan kuin se olisi irrallinen kantaluokan olio. Tämän alustuksen hoitavat aivan normaalit kantaluokan rakentajat.

Ainoaksi ongelmaksi jäävät rakentajien parametrit. Aliluokan rakentaja saa kyllä parametrinsa aivan normaaliin tapaan aliluokan oliota luotaessa, mutta ongelmana on, miten kantaluokan rakentajalle saadaan välitetyksi sen tarvitsemat parametrit. C++:n tarjoama ratkaisu on, että aliluokan rakentajan *alustuslistassa* "kutsutaan" kantaluokan rakentajaa ja välitetään sille tarvittavat parametrit. Tällä tavoin aliluokka voi itse päättää, millaisia parametreja sen kantaluokalle välitetään olion luomisen yhteydessä. Listaus 6.2 seuraavalla sivulla näyttää esimerkin periytymisestä ja rakentajien käytöstä.

Jos aliluokan rakentajan alustuslistassa *ei* kutsuta mitään kantaluokan rakentajaa, yrittää kääntäjä olla ystävällinen. Tällaisessa tilanteessa se kutsuu nimittäin kantaluokan *oletusrakentajaa,* joka ei tarvitse parametreja. Tämä aiheuttaa kuitenkin sen, että rakentajakutsun unohtuessa alustuslistasta olion kantaluokkaosa alustetaan oletusarvoonsa, joka tuskin on haluttu lopputulos! Onkin erittäin tärkeää, että aliluokan rakentajan alustuslistassa kutsutaan ***aina*** jotain kantaluokan rakentajaa.

Aliluokan olion alustusjärjestys C++:ssa on sellainen, että ensimmäisenä suoritetaan kantaluokan rakentaja kokonaisuudessaan, jonka jälkeen suoritetaan aliluokan oma rakentaja. Mikäli periytymishierarkia on korkeampi kuin kaksi luokkaa, lähdetään rakentajia suorittamaan aivan hierarkian huipusta lähtien alaspäin, niin että olio ikään kuin rakentuu vähitellen laajemmaksi ja laajemmaksi. Tämä alustusjärjestys takaa sen, että aliluokan rakentajassa voidaan jo turvallisesti kutsua kantaluokan jäsenfunktioita, koska aliluokan rakentajan koodiin päästäessä kantaluokkaosa on jo alustettu kuntoon (poikkeuksena ovat aliluokan uudelleenmäärittelemät *virtuaalifunktiot*, joiden käyttäytymistä rakentajien kanssa käsitellään tarkemmin aliluvus-

```
.................................. Kantaluokka ..................................
 1 class Lokiviesti
 2 {
 3 public:
 4   Lokiviesti(string const& viesti);
        ⋮
 5 private:
 6   string viesti_;
 7 };
 8
 9 Lokiviesti::Lokiviesti(string const& viesti) : viesti_(viesti)
10 {
11 }
.................................. Aliluokka ..................................
 1 class PaivattyLokiviesti : public Lokiviesti
 2 {
 3 public:
 4   PaivattyLokiviesti(Paivays const& pvm, string const& viesti);
        ⋮
 5 private:
 6   Paivays pvm_;
 7 };
 8
 9 PaivattyLokiviesti::PaivattyLokiviesti(Paivays const& pvm,
10   string const& viesti) : Lokiviesti(viesti), pvm_(pvm)
11 {
12 }
.............................. Olion luominen ..............................
 1   Lokiviesti viesti("Kävin leffassa");
 2   PaivattyLokiviesti pvmviesti(tanaan(), "Huono oli");
```

LISTAUS 6.2: Periytyminen ja rakentajat

sa 6.5.7).

6.3.3 Periytyminen ja purkajat

Alustamisen tapaan myös aliluokan olion siivoustoimenpiteet vaativat erikoiskohtelua luokan "kerrosrakenteen" vuoksi. Samoin kuin aliluokan olion alustaminen, myös sen siivoaminen on jaettu vastuualueiden kesken. Kantaluokan purkajan tehtävänä on siivota kantaluokkaolio sellaiseen kuntoon, että se voi rauhassa tuhoutua. Aliluokan purkajan vastuulla on vastaavasti pitää huoli siitä, että aliluokan

olion *periytymisessä lisätty laajennusosa* siivotaan tuhoutumiskuntoon.

Kuten aiemminkin, ohjelmoijan ei itse tarvitse huolehtia purkajien kutsumisesta vaan olion tuhoutuessa kääntäjä kutsuu automaattisesti tarvittavia purkajia. Periytymisen yhteydessä olion purkajien kutsujärjestys on päinvastainen rakentajien kutsujärjestykseen nähden eli ensin kutsutaan aliluokan purkajaa, sitten kantaluokan purkajaa ja tarvittaessa tämän kantaluokan purkajaa ja niin edelleen. Tämä kutsujärjestys varmistaa sen, että aliluokan purkajaa suoritettaessa olion kantaluokkaosa on vielä käyttökelpoinen, ja näin aliluokan purkaja voi vielä turvallisesti kutsua kantaluokan jäsenfunktioita (kuten rakentajissakin, aliluokan uudelleenmäärittelemät virtuaalifunktiot ovat poikkeustapaus. Niitä ja purkajia käsitellään tarkemmin aliluvussa 6.5.7).

Kantaluokan purkaja kannattaa määritellä lähes aina *virtuaaliseksi*, mutta tätä käsitellään vasta aliluvussa 6.5.5.

6.3.4 Aliluokan olion ja kantaluokan suhde

Koska aliluokka perii kantaluokan kaikki ominaisuudet, tarjoaa aliluokan olio ulospäin kaikki ne palvelut, jotka kantaluokan oliokin tarjoaa. Näin ollen aliluokan oliota voisi sen ulkoista rajapintaa ajatellen käyttää kaikkialla, missä kantaluokan oliotakin — periytymisessähän vain lisätään ominaisuuksia.

Tämä ajatus aliluokan olion kelpaamisesta kantaluokan olion paikalle on viety useissa oliokielissä vielä pitemmälle määrittelemällä, että kielen tyypityksen kannalta ***aliluokan olio on tyypiltään myös kantaluokan olio***. Näin aliluokan oliot kuuluvat ikään kuin useaan luokkaan: aliluokkaan itseensä, kantaluokkaan, mahdollisesti kantaluokan kantaluokkaan jne. Tämä *is-a* -suhde tulisi pitää mielessä aina, kun periytymistä käytetään. Jos aliluokka on muuttunut vastuualueeltaan niin paljon, että se ei enää ole kantaluokan mukainen, periytymistä on käytetty mitä ilmeisimmin väärin.

C++:n kannalta tämä ominaisuus tarkoittaa, että kielen tyypityksessä aliluokan olio kelpaa kaikkialle minne kantaluokan oliokin. Erityisesti kantaluokkaosoittimen tai -viitteen voi laittaa osoittamaan myös

aliluokan olioon:

```
class Kantaluokka { ... };
class Aliluokka : public Kantaluokka { ... };
void funktio(Kantaluokka& kantaolio);

Kantaluokka* k_p = 0;
Aliluokka aliolio;
k_p = &aliolio;
funktio(aliolio);
```

Tämä tilanne, jossa osoittimen päässä olevan olion tyyppi ei ole sama kuin osoittimen tyyppi, ei aiheuta yleensä ongelmia, koska jokainen aliluokan olio tarjoaa periytymisestä johtuen kaikki kantaluokan tarjoamat palvelut. Tämä mahdollisuus käyttää aliluokan olioita ohjelmassa kantaluokan sijaan on yksi tärkeimpiä olio-ohjelmoinnin ja periytymisen työkaluja.[ϒ] Sen käyttöä käsitellään aliluvussa 6.5.2.

6.4 C++: Periytymisen käyttö laajentamiseen

Periytymisen kenties yksinkertaisin käyttötarkoitus on olemassa olevan luokan laajentaminen. Siinä aliluokka laajentaa kantaluokan tarjoamia palveluita tarjoamalla kaikki kantaluokan tarjoamat palvelut identtisinä ja sen lisäksi vielä uusia palveluita. Tällainen periytymisen käyttö mahdollistaa koodin uudelleenkäytön, koska aliluokan ei tarvitse kirjoittaa uudelleen kantaluokan jo kertaalleen toteuttamia palveluita.

Listaus 6.3 seuraavalla sivulla näyttää yksinkertaisen luokan `Kirja`, joka muistaa yksittäisen kirjan tiedot. Periaatteessa listauksessa esitetty luokka sopisi muuten hyvin kirjaston kortistojärjestelmässä käytetyksi kirjaksi, mutta kirjaston kirjoilla on yksi olennainen lisäominaisuus: niillä on viimeinen palautuspäivämäärä. (Listauksen rivillä 5 on purkajan edessä avainsana **`virtual`**. Sillä ei ole tämän esimerkin kannalta merkitystä, mutta näin tehdään yleensä kaikissa kantaluokissa. Virtuaalipurkajan merkitys selitetään aliluvussa 6.5.5.)

[ϒ] C++:ssa vaaraksi muodostuu **viipaloituminen** (*slicing*). Kun esimerkiksi kantaluokan olioon sijoitetaan aliluokan olio (joka on tyypiltään myös kantaluokan olio), suoritetaan sijoituksessa vain kantaluokkaosan sijoitus ja aliluokkaosa jää sijoituksessa käyttämättä. Tätä käsitellään aliluvussa 7.1.3.

Luokan esittely

```
 1 class Kirja
 2 {
 3 public:
 4   Kirja(std::string const& nimi, std::string const& tekija);
 5   virtual ~Kirja();
 6   std::string annaNimi() const;
 7   std::string annaTekija() const;
        ⋮
11 private:
        ⋮
13   std::string nimi_;
14   std::string tekija_;
15 };
```

Luokan toteutus

```
 1 Kirja::Kirja(string const& n, string const& t) : nimi_(n), tekija_(t)
 2 {
 3   cout << "Kirja " << nimi_ << " luotu" << endl;
 4 }
 5
 6 Kirja::~Kirja()
 7 {
 8   cout << "Kirja " << nimi_ << " tuhottu" << endl;
 9 }
10
11 string Kirja::annaNimi() const
12 {
13   return nimi_;
14 }
        ⋮
```

LISTAUS 6.3: Kirjan tiedot muistava luokka

Jos olemassa olevaa kirjaluokkaa halutaan käyttää uuden luokan pohjana periytymisessä, on tärkeää ensin varmistua siitä, että uusi luokka tarjoaa sellaisenaan kaikki vanhan luokan palvelut ja lisää siihen lisäksi uusia palveluita. Kirjan ja kirjaston kirjan tapauksessa nämä edellytykset toteutuvat, koska kaikki kirjan tarjoamat palvelut sopivat sellaisenaan myös kirjaston kirjalle. Listauksessa 6.4 on luokasta `Kirja` periytetty uusi laajempi luokka `KirjastonKirja`, joka tarjoaa kaikki kirjan palvelut ja lisäksi palautuspäivämäärän käsittelyn.

Periyttämistä ei koskaan kannata käyttää turhaan, koska se monimutkaistaa ohjelmaa. Esimerkiksi yllä oleva kirjastonkirjan periyttäminen kirjasta on perustelua vain, jos jossain todella tarvitaan myös tavallista kirjaluokkaa tai jos kirjaluokka on saatu muualta. Mikäli

```
............................... Luokan esittely ...............................
 1 class KirjastonKirja : public Kirja
 2 {
 3 public:
 4   KirjastonKirja(std::string const& nimi, std::string const& tekija,
 5                  Paivays const& palpvm);
 6   virtual ~KirjastonKirja();
 7   bool onkoMyohassa(Paivays const& tanaan) const;
       ⋮
 9 private:
10   Paivays palpvm_;
11 };
.............................. Luokan toteutus ..............................
 1 KirjastonKirja::KirjastonKirja(string const& nimi, string const& tekija,
 2   Paivays const& palpvm) : Kirja(nimi, tekija), palpvm_(palpvm)
 3 {
 4   cout << "Kirjastonkirja " << nimi << " luotu" << endl;
 5 }
 6
 7 KirjastonKirja::~KirjastonKirja()
 8 {
 9   cout << "Kirjastonkirja " << annaNimi() << " tuhottu" << endl;
10 }
11
12 bool KirjastonKirja::onkoMyohassa(Paivays const& tanaan) const
13 {
14   return palpvm_.paljonkoEdella(tanaan) < 0;
15 }
```

LISTAUS 6.4: Kirjaston kirjan palvelut tarjoava aliluokka

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.

ISOC++: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.

Luvussa 6.3.2 selitettiin, kuinka aliluokan olion rakentaminen tapahtuu “kerroksittain” niin, että ensin suoritetaan kantaluokan rakentaja ja sitten aliluokan rakentaja. Tämä suoritusjärjestys aiheuttaa sen, että kantaluokan rakentajan koodissa olion “aliluokkaosaa” ei ole vielä alustettu, eikä se näin ollen ole käyttökunnossa. Asian voi ajatella niinkin, että ennen aliluokan rakentajan suorittamista luotava olio ei vielä ole aliluokan olio, vaan vasta kantaluokan olio. Kun sitten kantaluokan rakentaja on saatu suoritetuksi, olion tyyppi “täydentyy” aliluokan olioksi aliluokan rakentajaan siirryttäessä, kunnes lopulta kaikki rakentajat on saatu suoritetuksi ja koko olio alustetuksi.

Tällainen kerroksittainen rakentuminen vaikuttaa siihen, miten virtuaalifunktiot käyttäytyvät. Kantaluokan rakentajan koodissa olio ei vielä oikeastaan ole aliluokan olio eikä näin ollen pysty suorittamaan aliluokassa määriteltyä virtuaalifunktion toteutusta. Niinpä rakentajassa kutsuttu virtuaalifunktio käyttäytyykin ikään kuin olio olisi vain kantaluokan olio. Sen toteutusta ei etsitä alempaa luokkahierarkiasta, vaan ainoastaan itse kantaluokasta tai sen omista kantaluokista. Tämä ei yleensä ole se, mitä ohjelmoija haluaa, joten ***rakentajien koodissa ei yleensä pitäisi kutsua virtuaalifunktioita***, ellei ole aivan varma siitä, että haluaa kutsua nimenomaan kantaluokan omaa toteutusta.

Täsmälleen sama tilanne tapahtuu purkajissa, mutta päinvastaisista syistä. Kun aliluokan olio tuhotaan, kutsutaan ensin aliluokan omaa purkajaa, jonka tehtävänä on siivota aliluokan osa oliosta. Tämän jälkeen siirrytään kantaluokan purkajaan ja niin edelleen, kunnes päästään periytymishierarkian huipulle. Kantaluokan purkajan koodissa olio ei enää oikeastaan ole aliluokan olio, koska tämä osa oliosta on jo siivottu tuhoamiskuntoon. Niinpä kantaluokan purkajassa kutsutut virtuaalifunktiot käyttäytyvät samoin kuin kantaluokan rakentajissa, ja niitä kutsutaan aivan kuin olio olisi kantaluokan olio. Tämä tarkoittaa, että virtuaalifunktion toteutusta etsitään vain kantaluokasta itsestään tai sen omista kantaluokista. Näin myöskään ***purkajien koodissa ei yleensä pitäisi kutsua virtuaalifunktioita***.

6.6 Abstraktit kantaluokat

Aiemmin aliluvussa 6.1 mainittiin **abstraktit kantaluokat** ja määriteltiin ne luokkahierarkiassa oleviksi luokiksi, joiden ainoa merkitys on olla periytymisessä kantaluokkina ja joista ei ole mielekästä tehdä olioita. Abstraktin kantaluokan perustava ominaisuus on, että siinä määritellään rajapintafunktioita, joille määritellään toteutus vasta myöhemmin luokkahierarkiassa.

C++:ssa abstrakteja kantaluokkia voi määritellä käyttämällä **puhtaita virtuaalifunktioita** (*pure virtual function*). Puhtaalla virtuaalifunktiolla tarkoitetaan virtuaalifunktiota, *jonka toteutus on pakko määritellä aliluokissa*. Virtuaalifunktio tehdään puhdas virtuaalifunktio lisäämällä luokan esittelyssä sen jälkeen määre `=0`. Listaus 6.10 seuraavalla sivulla sisältää esimerkkinä luokkien Eläin ja Lintu esittelyt, joissa funktiot `liiku` ja `laula` on määritelty puhtaiksi virtuaalifunktioiksi.

Abstraktiksi kantaluokaksi määritellään C++:ssa mikä tahansa luokka, jossa on ainakin yksi puhdas virtuaalifunktio. Tämä funktio voi olla luokassa itsessään määritelty tai peritty kantaluokalta. Periytetyissä luokissa voidaan tällainen puhdas virtuaalifunktio sitten määritellä taas tavalliseksi virtuaalifunktioksi ja kirjoittaa sille kyseiseen luokkaan sopiva toteutus. Näin periytymisen yhteydessä puhtaat virtuaalifunktiot muuttuvat taas normaaleiksi virtuaalifunktioiksi.

Lopulta periytymishierarkian alapäässä ollaan tilanteessa, jossa luokissa ei enää ole yhtään puhdasta virtuaalifunktiota vaan kaikille kantaluokissa määritellyille puhtaille virtuaalifunktioille on olemassa toteutus. Tällaiset luokat ovat vihdoin “konkreettisia” luokkia, joista löytyy toteutus kaikille rajapintafunktioille ja joista on näin järkevää luoda oliota. Listauksen 6.10 esimerkissä luokka `Kana` määrittelee kaikki kantaluokkien puhtaat virtuaalifunktiot tavallisiksi virtuaalifunktioiksi ja tarjoaa niille toteutuksen, joten kanat edustavat luokkahierarkiassa “konkreettisia” olioita.

Kääntäjä pitää automaattisesti huolen siitä, että abstrakteista kantaluokista ei voi luoda oliota. Niiden käyttö ohjelmassa rajoittuukin siihen, että niitä voidaan käyttää periytymisessä kantaluokkana, ja siihen, että ohjelmassa voi olla osoittimia ja viitteitä tällaiseen kantaluokkaan. Näiden osoittimien ja viitteiden päähän voi sitten sijoittaa abstrakteista kantaluokista periytettyjen luokkien olioita ja kutsua ali-

```
11 class Elain : public Elio
12 {
13 public:
14   virtual ~Elain();
15   virtual void liiku(Sijainti paamaara) = 0;
18 };
        ⋮
19 class Lintu : public Elain
20 {
21 public:
22   virtual ~Lintu();
23   virtual void laula() = 0;
24 };
        ⋮
25 class Kana : public Lintu
26 {
27 public:
28   virtual ~Kana();
29   virtual void lisaanny(); // Toteutus lisääntymisfunktiolle
30   virtual void liiku(Sijainti paamaara); // Toteutus liikkumiselle
31   virtual void laula(); // Toteutus laulamiselle
32
33 private:
34   // Tänne tarvittava sisäinen toteutus
35 };
```

LISTAUS 6.10: Abstrakteja kantaluokkia ja puhtaita virtuaalifunktioita

luokissa määriteltyjä puhtaiden virtuaalifunktioiden toteutuksia dynaamista sitomista hyväksikäyttäen.

Vaikka abstrakti kantaluokka pakottaakin aliluokat kirjoittamaan omat toteutuksensa puhtaalle virtuaalifunktiolle, voi kantaluokka C++:ssa tästä huolimatta sisältää oman toteutuksensa tälle funktiolle. Tällaisessa tapauksessa voidaan ajatella, että kantaluokka sisältää *osittaisen* toteutuksen, joka toteuttaa rajapintafunktiosta sellaisen osan, joka on *yhteistä* kaikille tästä luokasta periytetyille luokille. Aliluokkien toteutukset voivat sitten kutsua tätä kantaluokan tarjoamaa toteutusta, ja tällä tavalla kaikissa aliluokissa toistuvaa yhteistä osaa ei tarvitse kopioida erikseen jokaiseen aliluokkaan. Puhtaalle virtuaalifunktiolle, jolla on toteutus, löytyy joskus myös muutakin käyttöä. Niistä löytyy tietoa esimerkiksi teoksesta “More Exceptional C++” [Sutter, 2002c].

Listauksessa 6.11 on esimerkki luokasta `Eläin`, joka sisältää jäsenmuuttujanaan tiedon eläimen sijainnista. Tässä luokassa määritellään myös "kaikille eläimille yhteinen" toteutus liikkumiselle, nimittäin eläimen sijainnin päivitys. Tämä toteutus ei kuitenkaan ole riittävä, koska liikkumiseen liittyy paljon muutakin (reitin valinta, lihasten liikuttaminen), joten aliluokissa on pakko määritellä "tarkempi" toteutus liikkumiselle. On huomattava, että vaikka abstrakti kantaluokka sisältäisikin toteutuksen puhtaalle virtuaalifunktiolle, konkreettisten aliluokkien on silti pakko määritellä funktiolle oma toteutuksensa.

C++:n abstraktit kantaluokat voivat puhtaiden virtuaalifunktioiden lisäksi sisältää mitä tahansa muutakin. Abstrakti kantaluokka voi määritellä kaikille aliluokille yhteisiä jäsenmuuttujia, uusia jäsenfunktioita ja niin edelleen. Se voi myös määritellä toteutuksia omien kantaluokkiensa määrittelemille puhtaille virtuaalifunktioille ja näin tarjota näille rajapintafunktiolle toteutuksen, joka periytyy myös kaikille aliluokille. Nämä aliluokat voivat puolestaan joko hyväksyä kan-

```
11 class Elain : public Elio
12 {
13 public:
14   virtual ~Elain();
15   virtual void liiku(Sijainti paamaara) = 0;
16 private:
17   Sijainti paikka_;
18 };
        ⋮
1 // Kaikille eläimille yhteinen osa liikkumista
2 void Elain::liiku(Sijainti paamaara)
3 {
4   paikka_ = paamaara;
5 }
        ⋮
1 // Kanan liikkuminen
2 void Kana::liiku(Sijainti paamaara)
3 {
4   // Tähän kanaan liittyvät erityistoimet, käveleminen jne.
5   Elain::liiku(paamaara); // Kantaluokka suorittaa kaikille yhteiset toimet
6 }
```

LISTAUS 6.11: Puhdas virtuaalifunktio, jolla on myös toteutus

taluokan tarjoaman toteutuksen tai kirjoittaa omansa. Näin abstraktien kantaluokkien käyttömahdollisuudet ovat varsin laajat.

6.7 Moniperiytyminen

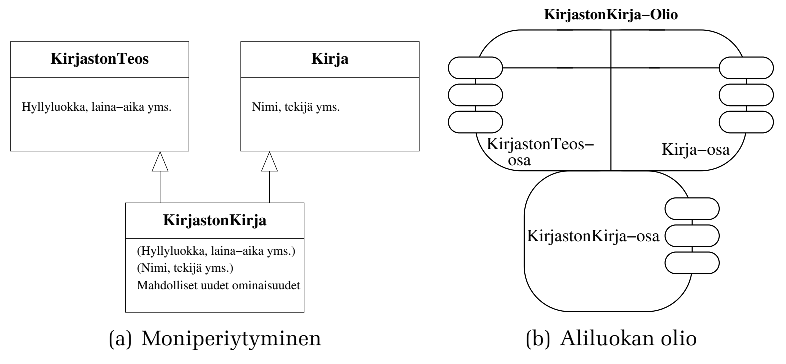
Periytymisessä uusi luokka luodaan periyttämällä siihen jonkin olemassa olevan luokan ominaisuudet. Mikään ei tietysti estä laajentamasta tätä mekanismia niin, että luokka periytetään *useasta* kantaluokasta. Tästä käytetään nimitystä **moniperiytyminen** (*multiple inheritance*).

Moniperiytyminen on yksinkertaisesta ideastaan huolimatta varsin kiistelty mekanismi, jonka käyttö oikein on joskus vaikeaa ja johon liittyy paljon piileviä vaaroja. Niinpä moniperiytymistä ei ole otettu mukaan läheskään kaikiin oliokieliin ainakaan täydellisesti toteutettuna. Tämän aliluvun tarkoituksena on antaa yleiskatsaus moniperiytymiseen C++:ssa ja antaa muutama esimerkki sen vaaroista ja hyötykäytöstä. On kuitenkin heti alkuun syytä varoittaa, että ***moniperiytymisen käyttöä tulisi yleensä välttää,*** koska se saattaa varsin usein johtaa ongelmiin. Sillä on kuitenkin oma paikkansa olio-ohjelmoinnissa, joten sen perusteet on syytä käsitellä.

6.7.1 Moniperiytymisen idea

Moniperiytymisessä luokka perii ominaisuudet useammalta kuin yhdeltä kantaluokalta. Tässä suhteessa moniperiytyminen ei tuo mitään uutta periytymiseen. Aliluokan olio koostuu normaalissa moniperiytymisessä kaikkien kantaluokkiensa kantaluokkaosista sekä aliluokan omasta laajennusosasta. Kuva 6.6 seuraavalla sivulla kuvaa tätä tilannetta. Niin sanotussa **yhdistävässä moniperiytymisessä** eli **virtuaalimoniperiytymisessä** (*shared multiple inheritance, virtual multiple inheritance*) tilanne on hiukan toinen, mutta sitä käsitellään vasta aliluvussa 6.8.2.

Tavallisessa periytymisessä aliluokan oliota voi aina ajatella myös kantaluokan oliona. Samalla tavalla moniperiytymisessä aliluokan olio kuuluu ikään kuin *yhtaikaa* kaikkiin kantaluokkiinsa. Toisin sanottuna aliluokan olio käy aina *minkä tahansa* kantaluokkansa edustajaksi. Kuvan 6.6 esimerkissä tämä toteutuu, koska kirjaston kirja on *aina* yhtaikaa sekä kirja että kirjaston teos. Tämä “aliluokan olio on



(a) Moniperiytyminen (b) Aliluokan olio

KUVA 6.6: Moniperiytyminen ja sen vaikutus

aina kaikkien kantaluokkiensa olio" -periaate on erittäin tärkeää moniperiytymisen oikean käytön kannalta.

6.7.2 Moniperiytyminen eri oliokielissä

Moniperiytyminen tuo mukanaan hyötyjen lisäksi monia ongelmia ja virhemahdollisuuksia. Niinpä eri oliokielten kehittäjät ovat suhtautuneet moniperiytymiseen varsin eri tavoin. Joissain kielissä moniperiytymistä ei ole ollenkaan (Smalltalk), joissain sen käyttöä on rajoitettu (Java) ja joissain (kuten C++:ssa) se on annettu ongelmineen kaikkineen ohjelmoijan käyttöön.

C++:n suhtautuminen moniperiytymiseen on yksi sallivimmista oliokielten joukossa. C++:ssa on pyritty antamaan ohjelmoijalle mahdollisuus käyttää moniperiytymistä juuri niin kuin ohjelmoija haluaa, ja samalla kaikki moniperiytymisen mukanaan tuomat riskit ja ongelmat on jätetty ohjelmoijan ratkaistavaksi. Tämän tuloksena C++:n moniperiytyminen on kyllä voimakas työkalu, mutta toisaalta sitä on erittäin helppo käyttää väärin niin, että sen aiheuttamat ongelmat havaitaan vasta liian myöhään. Moniperiytymisen käyttö C++:ssa vaatiikin ohjelmoijalta itsekuria ja olioajattelun hallitsemista.

Javassa varsinaista luokkien moniperiytymistä ei ole lainkaan. Jokaisella luokalla on vain yksi kantaluokka, josta se periytyy. Sen si-

jaan Java sallii erilliset **rajapintaluokat** (*interface class*), joita voi yhdistellä moniperiytymisen tapaan. Tämä tekee mahdolliseksi sen, että luokkien *rajapintoja* voi yhdistellä monesta luokasta vapaasti, sen sijaan *toteutuksen* moniperiytyminen ei ole mahdollista. Rajapintaluokkia käsitellään tarkemmin aliluvussa 6.9.

Smalltalk ottaa vielä Javaakin tiukemman kannan moniperiytymiseen. Tässä kielessä moniperiytymistä ei yksinkertaisesti ole. Jokaisella luokalla on tasan yksi välitön kantaluokka, eikä tähän voi vaikuttaa millään tavalla. Käytännössä Smalltalkin periytyminen sallii kuitenkin kaiken minkä Javankin. Smalltalkissa ei nimittäin ole tiukkaa käännösaikaista tyypitystä, vaan esimerkiksi kielen olioviitteet voivat aina viitata mihin tahansa olioon sen luokasta riippumatta ja mille tahansa oliolle voi yrittää kutsua mitä tahansa palvelua. Tästä johtuen Smalltalkissa voi helposti tehdä usealle eri luokalle toimivaa koodia, vaikkei luokilla olisikaan yhteistä rajapintaa määräävää kantaluokkaa.

6.7.3 Moniperiytymisen käyttökohteita

Kiistanalaisuudestaan huolimatta moniperiytymiselle on muutamia selkeitä käyttökohteita, joissa siitä on hyötyä. Heti alkuun on kuitenkin syytä huomauttaa, että missään seuraavassa esiteltävistä tilanteista moniperiytyminen ei ole *aivan* välttämätön. Kaikki esitetyt tilanteet voidaan aina ratkaista myös ilman moniperiytymistä — kylläkin usein hieman työläämmin ja monimutkaisemmin (samaan tapaan kuin kaikki ohjelmat on mahdollista koodata ilman koko olio-ohjelmointia).

Ehkä tyypillisin ja turvallisin käyttökohde moniperiytymiselle on *rajapintojen yhdistäminen.* Jos sama luokka toteuttaa kerralla useita (yleensä abstraktien) kantaluokkien määräämiä rajapintoja, tekee moniperiytyminen tämän ilmaisemisen helpoksi. Luokka moniperiytetään kaikista niistä kantaluokista (**rajapintaluokista**), joiden rajapinnat se toteuttaa, ja puhtaiden virtuaalifunktioiden toteutukset kirjoitetaan aliluokkaan. Tämä moniperiytymisen muoto on juuri se, jota Java-kieli tukee, vaikkei siinä täydellistä moniperiytymistä olekaan. Moniperiytymisen käyttö tähän tarkoitukseen on suhteellisen mutkatonta ja varsin hyödyllistä, joten sitä käsitellään tarkemmin aliluvussa 6.9.

Toinen käyttökohde moniperiytymiselle on kahden tai useamman olemassa olevan luokan yhdistäminen. Varsin monet valmiit oliokirjastot sisältävät luokkia, jotka on suunniteltu käytettäväksi kantaluokkina, joita ohjelmoija laajentaa ja tarkentaa periyttämällä niistä omat luokkansa. Tämä pätee erityisesti sovelluskehyksissä, jotka esitellään lyhyesti aliluvussa 6.11. Jos nyt ohjelmoijalle tulee tarve kirjoittaa luokka, jonka selkeästi pitäisi laajentaa tai tarkentaa useaa valmista kantaluokkaa, on moniperiytyminen usein ainoa käytännöllinen vaihtoehto. Tällaisessa tilanteessa on kuitenkin syytä olla tarkkana, koska joissain tapauksissa moniperiytettävät kantaluokat saattavat häiritä toistensa toimintaa. Tällaisiin tilanteisiin tutustutaan aliluvuissa 6.8.1 ja 6.8.2.

Kolmantena moniperiytymistä käytetään joskus luokkien koostamiseen valmiista ominaisuuskokoelmista. Tästä käytetään englanninkielisessä kirjallisuudessa usein nimitystä *mixin* tai *flavours*.[♊] Esimerkkinä tällaisesta periytymisestä voisi olla tilanne, jossa kaikki lainaamiseen ja palautuspäivämäärään liittyvät rajapinnat ja toiminnallisuus on kirjoitettu erilliseen luokkaan Lainattava. Vastaavasti tuotteen myymiseen liittyvät asiat ovat luokassa Myytävä. Näiden avulla luokka KirjastonKirja voitaisiin luoda moniperiyttämällä peruskantaluokka Kirja ja ominaisuusluokka Lainattava. Samoin kaupan oleva CD-ROM saataisiin moniperiyttämällä luokat Cdrom ja Myytävä.

Edellä esitettyjä moniperiytymisen käyttötapoja on vielä mahdollisuus yhdistää. On esimerkiksi mahdollista tehdä erillinen rajapinta ja useita sen eri tavalla toteuttavia ominaisuusluokkia. Näitä käyttäen ohjelmoija voi moniperiyttämällä ilmoittaa oman luokkansa toteuttavan tietyn rajapinnan ja vielä periyttää mukaan valitsemansa toteutuksen. Tällainen moniperiytymisen käyttö ei kuitenkaan ole enää aivan yksinkertaista, eikä sitä käsitellä tässä tarkemmin.

6.7.4 Moniperiytymisen vaaroja

Suuri osa moniperiytymisen vaaroista johtuu siitä, että moniperiytyminen on houkuttelevan helpontuntuinen vaihtoehto sellaisissakin tapauksissa, joissa se ei olioajattelun kannalta ole perusteltua. Moni-

[♊] Englanninkieliset termit ovat lähtöisin amerikkalaisista jäätelöbaareista, joissa asiakkaat saavat usein itse päättää jäätelönsä maun, ja jäätelö valmistetaan paikan päällä sekoittamalla (mixing) vaniljajäätelöön marjoja, suklaata, pähkinöitä tai muita makuaineita (flavours). Tästä syystä oliokirjallisuudessa "peruskantaluokasta" käytetään joskus nimitystä *vanilla*.

periytyminenkin on periytymistä, joten periytetyn luokan olion täytyy kaikissa tilanteissa olla käsitteellisesti myös kaikkien kantaluokkiensa olio. Moniperiytymistä *ei* voi esimerkiksi käyttää siihen, että aliluokan olio olisi *välillä* yhden, välillä toisen kantaluokan edustaja. Vaikkapa luokkaa Vesitaso ei voisi todennäköisesti moniperiyttää luokista Lentokone ja Vene, koska vesitaso ei kykene *yhtaikaa* liikkumaan sekä ilmassa että vedessä, toisin sanoen ilmassa ollessaan se ei ole vene ja vastaavasti vedessä liikkuessaan se ei ole lentokone.

Moniperiytyminen aiheuttaa normaalin periytymisen tapaan myös sen, että kaikkien kantaluokkien julkiset rajapinnat näkyvät aliluokasta ulospäin (poislukien C++:n yksityinen ja suojattu periytymistapa, jotka eivät ole perinteisen olioajattelun mukaisia). Niinpä moniperiytymistä ei tule käyttää tilanteissa, joissa halutaan saada luokkaan mukaan vain jonkin toisen luokan *toiminnallisuus*, muttei julkista rajapintaa. Esimerkiksi vaikka Paivays-luokka sisältääkin palautuspäivämäärän käsittelyyn tarvittavan toiminnallisuuden, ei luokkaa KirjastonKirja voi moniperiyttää luokista Kirja ja Paivays. Vaikka kirjaston kirja sisältääkin palautuspäivämäärän, *se ei ole päiväys,* eikä sen julkinen rajapinta sisällä Päiväys-luokan operaatioita. Tässä Päiväys-jäsenmuuttuja on selkeästi oikea ratkaisu.

Vaikka moniperiytymistä käytettäisiinkin olioajattelun kannalta oikein, se tekee ohjelman luokkarakenteesta helposti vaikeaselkoisen. Lisäksi moniperiytyminen aiheuttaa helposti ongelmia kuten rajapintojen moniselitteisyyttä ja vaikeuksia olion elinkaaren hallinnassa. Näitä ongelmia ja niiden ratkaisemista käsitellään lyhyesti myöhemmissä aliluvuissa. Kaikki tämä aiheuttaa kuitenkin sen, että moniperiytymistä ei kannata käyttää, ellei sille ole painavia perusteita.

6.7.5 Vaihtoehtoja moniperiytymiselle

Jos moniperiytymistä käytetään todella mallintamaan sitä, että aliluokan olio pysyvästi kuuluu useaan kantaluokkaan ja toteuttaa niiden rajapinnat, ei moniperiytymiselle ole helppoa vaihtoehtoa. Tällöin on kuitenkin usein kyse rajapintaluokkien periyttämisestä, joka ei juurikaan aiheuta ongelmia ja joka onnistuu myös esimerkiksi Javassa, jossa varsinaista moniperiytymistä ei ole.

Mikäli tarve moniperiytymiseen sen sijaan tulee siitä, että halutaan yhdistellä jo olemassa olevien luokkien ominaisuuksia, voi moniperiytymisen usein kiertää. Tällöin on nimittäin usein mahdollista

periytymisen sijaan käyttää koostetta eli ottaa valmiit luokat kirjoitettavan luokan jäsenmuuttujiksi. Tällöin valmiiden luokkien jäsenfunktiot eivät kuitenkaan näy ulospäin. Jos näitä jäsenfunktioita pitäisi pystyä kutsumaan kirjoitettavan luokan ulkopuolelta, täytyy tähän luokkaan vielä kirjoittaa "**läpikutsufunktiot**" (*call-through function*). Näillä tarkoitetaan jäsenfunktioita, jotka yksinkertaisesti kutsuvat vastaavaa jäsenmuuttujan jäsenfunktiota, välittävät sille saamansa parametrit ja palauttavat jäsenfunktiolta saamansa paluuarvon kutsujalle.

Moniperiytymisen korvaaminen koostamisella vaatii jonkin verran käsityötä, mutta sillä vältetään yleensä moniperiytymisen mukanaan tuomat ongelmat ja vaarat. Lisäksi se on ainoa vaihtoehto kielissä, joissa moniperiytymistä ei ole. On ehkä vielä syytä korostaa, että koostaminen ei käy vaihtoehdoksi moniperiytymiselle, jos luokkien välinen "is-a"-periytymissuhde on välttämätön esimerkiksi sen takia, että uutta luokkaa on pystyttävä käsittelemään kantaluokkaosoittimien kautta.

6.8 C++: Moniperiytyminen

C++:ssa moniperiytyminen tehdään yksinkertaisesti luettelemalla luokan esittelyn yhteydessä kaksoispisteen jälkeen kaikki luokan kantaluokat ja niiden periytymistavat (siis käytännössä lähes aina **public**, kuten aliluvussa 6.3 todettiin). Kuvan 6.6 luokka `KirjastonKirja` esiteltäisiin seuraavasti:

```
class KirjastonKirja : public KirjastonTeos, public Kirja
{
  // Tänne Kirjastonkirjan uudet lisäominaisuudet
};
```

Ominaisuuksiltaan moniperiytyminen ei C++:ssa eroa mitenkään normaalista periytymisestä. Kantaluokilta periytyneiden osien näkyvyydet ja muut ominaisuudet toimivat samoin kuin tavallisessa periytymisessäkin. Sama kantaluokka ei voi kuitenkaan esiintyä periytymislistassa kahteen kertaan. Toisin sanoen aliluokkaa ei voi suoraan periyttää "kahteen kertaan" samasta kantaluokasta. Epäsuorasti tämä kuitenkin onnistuu, jos kahdella eri kantaluokalla on keskenään

yhteisiä kantaluokkia. Tällainen tilanne johtaa kuitenkin helposti ongelmiin, ja sitä käsitellään myöhemmin tarkemmin toistuvan moniperiytymisen yhteydessä (aliluku 6.8.2).

Koska kantaluokkia on nyt useita, täytyy aliluokan rakentajan alustuslistassa luonnollisesti kutsua kaikkien kantaluokkien rakentajia. Muuten moniperiytyminen ei tuo ongelmia olioiden elinkaareen (poislukien jälleen pahamaineinen toistuva moniperiytyminen). Olion tuhoutumisen yhteydessä kaikkia tarvittavia purkajia kutsutaan edelleen automaattisesti.

6.8.1 Moniperiytyminen ja moniselitteisyys

Moniperiytyminen tuo joitain lisäongelmia verrattuna normaaliin periytymiseen. Kun kantaluokkia on useita, saattaa tietysti käydä niin, että samanniminen jäsenfunktio periytyy aliluokan rajapintaan useasta kantaluokasta. Tällöin ongelmaksi tulee päättää, minkä kantaluokan jäsenfunktiota tulisi kutsua, kun jäsenfunktiota kutsutaan aliluokan oliolle. C++:ssa tämä ongelma tulee esiin, kunhan jäsenfunktio nimi vain on sama, vaikka parametreissa olisikin eroa. Tällainen tilanne voisi esimerkiksi ilmetä, jos molemmat luokista KirjastonTeos ja Kirja määrittelisivät jäsenfunktion `tulostaTiedot`.

Tämä tilanne on ongelmallinen jo olioajattelunkin kannalta. Aliluokan olion tulisi rajapinnaltaan olla kaikkien kantaluokkiensa olio. Jos nyt vaikka päätettäisiin, että periytymislistalla ensimmäinen kantaluokka “voittaa” ja sen jäsenfunktio valitaan, oltaisiin ristiriidassa sen kanssa, että aliluokan olion tulisi olla toistenkin kantaluokkien olio, ja niissä tämä jäsenfunktio on erilainen. C++ ratkaisee ongelman niin, että yritys kutsua kahdesta eri kantaluokasta periytynyttä jäsenfunktiota aiheuttaa käännösaikaisen virheilmoituksen siitä, että jäsenfunktion kutsu on **moniselitteinen** (*ambiguous*).

Joskus eri kantaluokkien samannimiset jäsenfunktiot tekevät samantyyppisiä asioita (mihin samalla tavalla nimetty jäsenfunktio tietysti saattaa viitata), ja aliluokan toteutuksen tulisi suorittaa *kaikkien* kantaluokkien toteutus kyseisestä jäsenfunktiosta. Tämä onnistuu kohtalaisen helposti, jos kyseinen jäsenfunktio on *kaikissa* kantaluokissa virtuaalinen, jolloin aliluokka voi antaa sille oman toteutuksen. Aliluokka voi nyt omassa toteutuksessaan kutsua vuorollaan kaikkien kantaluokkien toteutusta jäsenfunktiosta ja kenties vielä lisätä jäsenfunktioon omaa toiminnallisuuttaan.

Kuten jo mainittiin, tällainen ratkaisu on perusteltu vain, jos kantaluokkien määrittelyt ja toteutukset jäsenfunktiolle eivät ole millään lailla ristiriidassa. Esimerkiksi KirjastonKirja-luokan `tulostaTiedot`-jäsenfunktion voisi kenties toteuttaa listauksen 6.12 esittämällä tavalla. Ikävä kyllä, käytännössä moniselitteisyyttä ei usein voi ratkaista näin helposti, mikä vähentää moniperiytymisen käyttömahdollisuuksia.

Jos moniselitteinen jäsenfunktio tosiaan tekee erilaisia, keskenään epäyhteensopivia asioita eri kantaluokissa, ei ole mitään mahdollisuutta saada aliluokan oliota käyttäytymään tämän jäsenfunktion osalta molempia kantaluokkia tyydyttävällä tavalla. Jos moniperiy-

```
 4 class KirjastonTeos
 5 {
 6 public:
 7   virtual void tulostaTiedot(std::ostream& virta) const;
         ⋮
 8 };
 9
10 class Kirja
11 {
12 public:
13   virtual void tulostaTiedot(std::ostream& virta) const;
         ⋮
14 };
15
16 class KirjastonKirja : public KirjastonTeos, public Kirja
17 {
18 public:
19   virtual void tulostaTiedot(std::ostream& virta) const;
         ⋮
20 };
         ⋮
21 void KirjastonKirja::tulostaTiedot(std::ostream& virta) const
22 {
23   Kirja::tulostaTiedot(virta);
24   KirjastonTeos::tulostaTiedot(virta);
25   // Tänne mahdollisesti vielä lisää tulostusta
26 }
```

LISTAUS 6.12: Moniselitteisyyden yksi välttämistapa

tymistä kuitenkin halutaan käyttää, voi tätä ongelmaa ratkaista erilaisilla tekniikoilla riippuen siitä, ovatko jäsenfunktiot virtuaalisia vai eivät. Seuraavassa esitetään muutaman tällaisen tekniikan perusteet. Tarkemmin niitä on käsitelty esimerkiksi kirjoissa "Effective C++" [Meyers, 1998] ja "More Exceptional C++" [Sutter, 2002c].

Mikäli moniselitteisille jäsenfunktioille ei ole tarkoitus antaa uusia toteutuksia moniperiytetyssä aliluokassa, muodostuu ainoaksi ongelmaksi jäsenfunktion kutsuminen. Tämäkin on ongelma vain, kun jäsenfunktiota kutsutaan suoraan aliluokan rajapinnan kautta — siis suoraan oliota käyttäen tai aliluokkatyyppisen osoittimen tai viitteen kautta. Kantaluokkaosoittimienhan kautta moniselitteisyyttä ei ole, koska kullakin kantaluokalla on vain yksi mahdollinen toteutus jäsenfunktiolle. Tällaisessa tilanteessa ratkaisuvaihtoehtoja on kolme:

- Kutsutaan moniselitteistä jäsenfunktiota aina kantaluokkaosoittimien kautta — tarvittaessa vaikkapa väliaikaisia osoitinmuuttujia käyttäen. Tämä on helpoin mutta ehkä kömpelöin ratkaisu.

- Moniselitteisen jäsenfunktion kutsun yhteydessä on mahdollista erikseen kertoa, minkä kantaluokan versiota halutaan kutsua. Tämä onnistuu `::`-syntaksilla. Esimerkiksi `Kirja`-luokan `tulostaTiedot`-jäsenfunktiota voi kutsua syntaksilla

  ```
  KirjastonKirja k;
  k.Kirja::tulostaTiedot();
  ```

  Tämä syntaksi on kuitenkin myös ehkä hieman oudon näköinen ja vaatii kantaluokan nimen kirjoittamista näkyviin kutsun yhteyteen.

- Kolmas vaihtoehto on kirjoittaa aliluokkaan *uudet* keskenään erinimiset jäsenfunktiot, jotka kutsuvat kunkin kantaluokan toteutusta moniselitteiselle jäsenfunktiolle `::`-syntaksilla. Tällöin aliluokan rajapintaan täytyy dokumentoida tarkasti, että kyseiset jäsenfunktiot vastaavat kantaluokkien toisennimisiä jäsenfunktiota. Tällainen rajapinnan osittainen uudelleennimeäminen on joskus tarpeen. Listaus 6.13 seuraavalla sivulla näyttää esimerkin tästä ratkaisusta.

```
 1 class KirjastonKirja : public KirjastonTeos, public Kirja
 2 {
 3 public:
        ⋮
15   void tulostaKTeostiedot(std::ostream& virta) const;
16   void tulostaKirjatiedot(std::ostream& virta) const;
17 };
        ⋮
18 void KirjastonKirja::tulostaKTeostiedot(std::ostream& virta) const
19 {
20   KirjastonTeos::tulostaTiedot(virta);
21 }
22
23 void KirjastonKirja::tulostaKirjatiedot(std::ostream& virta) const
24 {
25   Kirja::tulostaTiedot(virta);
26 }
```

LISTAUS 6.13: Jäsenfunktiokutsun moniselitteisyyden eliminointi

Joskus moniperiytetyssä aliluokassa olisi tarpeen toteuttaa kultakin kantaluokalta periytynyt samanniminen (ja näin moniselitteinen) jäsenfunktio niin, että toteutus olisi *erilainen* kullekin kantaluokalle. Toisin sanoen kaksi kantaluokkaa tai useampi haluaa periytetyn luokan tarjoavan toteutuksen samannimisille jäsenfunktioille (samoilla parametreilla), mutta itse toteutuksien pitäisi olla keskenään erilaisia. Tällaisen pulman ratkaiseminen vaatii vähän lisäkikkailua. Ongelman voi ratkaista lisäämällä hierarkiaan kaksi abstraktia välikantaluokkaa, jotka "uudelleennimeävät" ongelmallisen jäsenfunktion. Tämä tapahtuu niin, että väliluokat määrittelevät keskenään erinimiset puhtaat virtuaalifunktiot, joita ne sitten kutsuvat ongelmallisen jäsenfunktion toteutuksessa. Lopullinen aliluokka toteuttaa nämä uudet jäsenfunktiot haluamallaan tavalla. Näitä toteutuksia voi sitten kutsua uusilla nimillä aliluokan rajapinnan kautta ja normaalisti alkuperäisellä nimellä kantaluokkaosoittimen tai -viitteen kautta. Listaus 6.14 seuraavalla sivulla näyttää esimerkin tästä tekniikasta.

C++:ssa on myös mahdollista määrätä aliluokka käyttämään yhden kantaluokan toteutusta **using**-määreen avulla. Tämä *ei* kuitenkaan ole olioajattelun kannalta oikea ratkaisu, koska tällöin dynaaminen sitominen ei tapahdu halutulla tavalla ja eri kantaluok-

```
 1 class KirjastonTeosApu : public KirjastonTeos
 2 {
 3 public:
 4   // Rakentaja parametrien välittämiseksi kantaluokan rakentajalle
 5   virtual void tulostaTiedot(std::ostream& virta) const;
 6   virtual void tulostaKTeostiedot(std::ostream& virta) const = 0;
 7 };
 8
 9 void KirjastonTeosApu::tulostaTiedot(std::ostream& virta) const
10 {
11   tulostaKTeostiedot(virta); // Kutsutaan aliluokan toteutusta
12 }
13
14
15 class KirjaApu : public Kirja
16 {
17 public:
29   // Rakentaja parametrien välittämiseksi kantaluokan rakentajalle
30   virtual void tulostaTiedot(std::ostream& virta) const;
31   virtual void tulostaKirjatiedot(std::ostream& virta) const = 0;
32 };
33
34 void KirjaApu::tulostaTiedot(std::ostream& virta) const
35 {
36   tulostaKirjatiedot(virta); // Kutsutaan aliluokan toteutusta
37 }
38
39
40 class KirjastonKirja : public KirjastonTeosApu, public KirjaApu
41 {
42 public:
43   virtual void tulostaKTeostiedot(std::ostream& virta) const;
44   virtual void tulostaKirjatiedot(std::ostream& virta) const;
45 };
46
47 void KirjastonKirja::tulostaKTeostiedot(std::ostream& virta) const
48 {
49   // Tänne KirjastonTeos-luokalle sopiva toteutus
50 }
51
52 void KirjastonKirja::tulostaKirjatiedot(std::ostream& virta) const
53 {
54   // Tänne Kirja-luokalle sopiva toteutus
55 }
```

LISTAUS 6.14: Moniselitteisyyden eliminointi väliluokilla

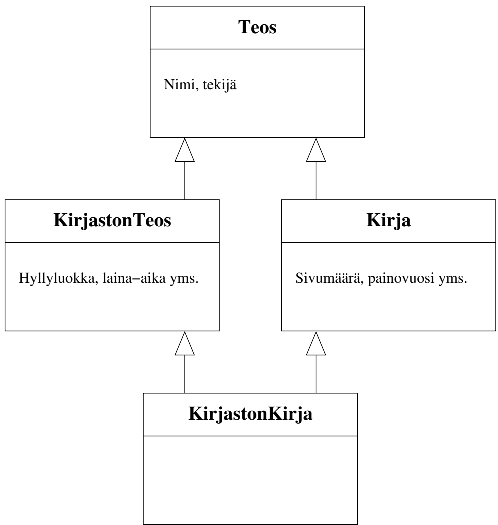
kaosoittimien kautta kutsutaan eri toteutuksia moniselitteiselle jäsenfunktiolle. Esimerkin tapauksessa luokan KirjastonKirja esittelyyn lisättäisiin jäsenfunktion `tulostaTiedot` esittelyn sijaan rivi "`using Kirja::tulostaTiedot;`", joka valitsee Kirja-luokan toteutuksen käyttöön. Tällöin kuitenkin edelleen KirjastonTeos-osoittimen läpi kutsuttaessa käytettäisiin KirjastonTeos-luokan toteutusta, mikä ei ole oikein.

6.8.2 Toistuva moniperiytyminen

Moniperiytyminen ei salli sitä, että sama kantaluokka esiintyisi periytymisessä suoraan kahteen kertaan. Tästä huolimatta kantaluokka voi silti periytyä mukaan useaan kertaan *epäsuorasti* välissä olevien kantaluokkien kautta. Kuva 6.7 seuraavalla sivulla näyttää tilanteen, jossa kantaluokka Teos päätyy aliluokkaan KirjastonKirja kahta reittiä kantaluokkien Kirja ja KirjastonTeos kautta. Tällaisesta käytetään usein termiä **toistuva moniperiytyminen** [Koskimies, 2000] (*repeated multiple inheritance* [Meyer, 1997]).

Tilanne, jossa kantaluokka periytyy aliluokkaan useaa eri reittiä, on hieman ongelmallinen. Se nimittäin herättää kysymyksen siitä, millainen aliluokan olion rakenteen tulisi olla. Jokaisessa KirjastonKirja-oliossa täytyy selvästi olla tyyppejä Kirja ja KirjastonTeos olevat kantaluokkaosat. Kummassakin kantaluokan oliossa puolestaan täytyy olla tyyppiä Teos oleva kantaluokkaosa. Tästä herää kysymys, onko KirjastonKirja-oliossa näitä Teos-tyyppisiä kantaluokkaosia kaksi (yksi kummallekin välittömälle kantaluokalle) vai yksi. Kuva 6.8 sivulla 188 havainnollistaa näitä vaihtoehtoja.

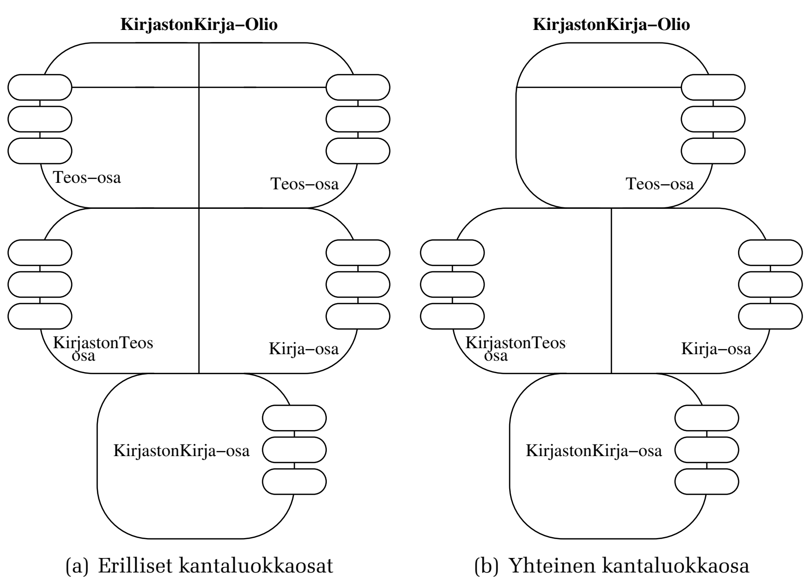
Kummassakin vaihtoehdossa on omat loogiset perustelunsa. Jos Teos-kantaluokkaosia on kaksi, muodostuu KirjastonKirja-olio selvästi kahdesta erillisestä kantaluokkaosasta. Tällöin puhutaan **erottelevasta moniperiytymisestä** (*replicated multiple inheritance*). Tässä tavassa on se hyvä puoli, että yhdessä kantaluokkaosassa tehdyt muutokset eivät mitenkään vaikuta toiseen kantaluokkaosaan ja kantaluokkaosat eivät häiritse toisiaan. Lisäksi tämä vaihtoehto on mahdollista toteuttaa ilman, että olion muistinkulutus tai tehokkuus kärsii mitenkään. Niinpä C++:n "normaalissa" moniperiytymisessä kantaluokkaosia voi olla tällä tavoin epäsuorasti useita kappaleita. Joissain tapauksissa tämä on myös selkeästi "oikea" vaihtoehto.



KUVA 6.7: Toistuva moniperiytyminen

Usean kantaluokkaosan vaihtoehdossa on kuitenkin haittana se, että läheskään aina — ja varsinkaan esimerkin tapauksessa — ei ole järkevää, että KirjastonKirja-oliossa on ikään kuin kaksi *erillistä* Teos-oliota. Kirjaston kirjahan on kaikesta huolimatta nimenomaan yksi ainoa Teos! Erottelevassa moniperiytymisessä kirjaston kirjan nimi ja tekijä talletetaan turhaan kahteen kertaan. Jos kirjaston kirjan nimeä nyt vielä muutetaan vain Kirja-luokan rajapinnan kautta, sisältävät Teos-kantaluokkaosat eri nimet, mikä ei tietenkään ole suotavaa. Kaiken kukkuraksi Teos-osoitinta ei voi laittaa osoittamaan KirjastonKirja-olioon, koska kääntäjä ei voi tietää, *kumpaan* kantaluokkaosaan osoittimen pitäisi osoittaa. Tässä syntyy siis samantapainen moniselitteisyys kuin aiemmin jäsenfunktioiden tapauksessa.

C++:ssa kahden kantaluokkaosan ongelma on ratkaistu niin, että periytymisen yhteydessä luokka voi "antaa luvan" siihen, että tarvittaessa moniperiytymisen yhteydessä muut aliolion osat voivat pitää tätä kantaluokkaosaa yhteisenä. Tätä sanotaan **virtuaaliseksi moniperiytymiseksi** (*virtual multiple inheritance*) tai **yhdistäväksi moniperiytymiseksi** (*shared multiple inheritance*), ja se merkitään C++:n periyty-



(a) Erilliset kantaluokkaosat (b) Yhteinen kantaluokkaosa

KUVA 6.8: Toistuva moniperiytyminen ja olion rakenne

misen yhteydessä avainsanalla **`virtual`**. Jos esimerkissä luokka `Kirja` on sallinut kantaluokan `Teos` jakamisen syntaksilla

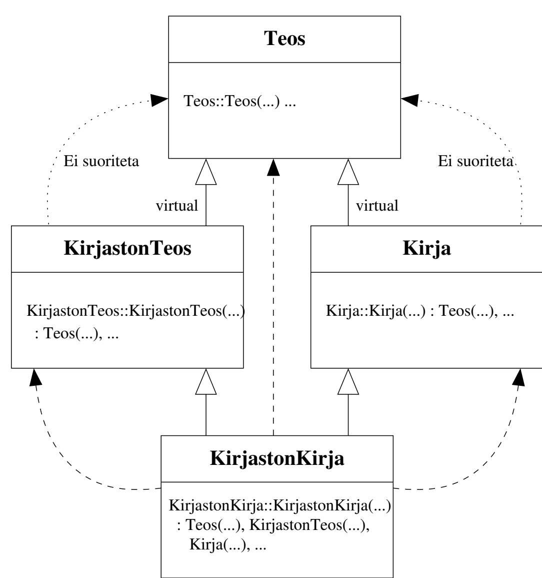
```
class Kirja : public virtual Teos // Tai virtual public...
```

ja luokka `KirjastonTeos` on tehnyt saman, tulee KirjastonKirja-olioon vain yksi Teos-kantaluokkaosa. Moniperiytymisen yhteydessä siis kaikki luokat, jotka on periytetty virtuaaliperiytymisellä samasta kantaluokasta, jakavat keskenään tämän kantaluokkaosan.

Virtuaaliperiytyminen tuo mukanaan yhden mutkan aliluokan olion elinkaareen. Aliluokan olion luomisen yhteydessä kutsutaan aliluokan rakentajaa, joka kutsuu kantaluokan rakentajaa ja niin edelleen. Virtuaaliperiytymisessä tämä tarkoittaisi, että jaetun kantaluokkaosan rakentaja suoritettaisiin *useita kertoja*, koska sitä luonnollisesti kutsutaan kaikkien siitä periytettyjen luokkien rakentajissa. Tämä ei tietenkään ole järkevää, koska yksi olio voidaan alustaa vain kertaalleen.

C++ ratkaisee tämän ongelman niin, että virtuaalisesti periytetyn kantaluokan rakentajaa täytyy kutsua suoraan sen luokan rakentajassa, jonka tyyppistä oliota ollaan luomassa, ja muut kantaluokkien rakentajissa olevat virtuaalisen kantaluokan rakentajakutsut jätetään suorittamatta. Esimerkin tapauksessa tämä tarkoittaa sitä, että `Teos`-luokan rakentajaa pitää kutsua suoraan `KirjastonKirja`-luokan rakentajan alustuslistassa, vaikka `Teos` ei olekaan tämän luokan välitön kantaluokka. Kun luokan `KirjastonKirja` oliota luodaan, `Teos`-luokan rakentajan kutsut luokissa `KirjastonTeos` ja `Kirja` jätetään suorittamatta. Näin yhteinen kantaluokkaosa alustetaan vain kertaalleen suoraan "alimman tason" luokan rakentajassa. Kuva 6.9 selventää tilannetta. Jos yhteisen kantaluokkaosan rakentajan kutsu puuttuu alimman tason rakentajasta, yrittää C++ tyypilliseen tapaansa kutsua kantaluokan oletusrakentajaa.

Käytännössä tämä vaatimus kutsua jaetun kantaluokan rakentajaa alimmassa aliluokassa "yli periytymishierarkian" hankaloittaa virtu-



KUVA 6.9: Rakentajat virtuaalisessa moniperiytymisessä

aalisen moniperiytymisen käyttöä ja tekee luokkien välisen vastuunjaon vaikeammaksi. Se on kuitenkin välttämätön rajoitus olioajattelun kannalta. Jos kaksi toisistaan tietämätöntä kantaluokkaa joutuu jakamaan yhteisen kantaluokkaosan, on luonnollista että nämä luokat moniperiyttävä aliluokka määrää, miten tämä yhteinen kantaluokka alustetaan. Helpommaksi tilanne muuttuu, jos jaetulla kantaluokalla on vain oletusrakentaja, jolloin rakentajan parametreista ei tarvitse välittää aliluokissa.

Olioiden tuhoamisen yhteydessä vastaavia ongelmia ei tule, vaan jaetun kantaluokan purkajaa kutsutaan normaalisti kertaalleen ilman, että aliluokkien täytyy ottaa siihen kantaa.[☺]

Kaiken kaikkiaan toistuva kantaluokka aiheuttaa moniperiytymiseen niin paljon monimutkaisuutta, rajoituksia ja ongelmia, että jotkut ovat antaneet tällaiselle periytymishierarkialle osuvan nimen "*Dreaded Diamond of Death*". Varsinkin virtuaalista moniperiytymistä pitäisikin yleensä välttää, ellei tiedä tarkkaan mitä on tekemässä. Joskus se on kuitenkin kelvollinen apukeino ohjelmoijan työkalupakissa.

6.9 Periytyminen ja rajapintaluokat

On varsin yleistä, että abstrakteissa kantaluokissa määritellään luvun alun eliöesimerkin tapaan pelkästään puhtaita virtuaalifunktioita, jolloin abstraktit kantaluokat eivät sisällä mitään muuta kuin rajapinnan määrittelyjä. Tällöin puhutaan usein **rajapintaluokista** (*interface class*). Rajapintaluokissa ei siis ole jäsenmuuttujia eikä jäsenfunktioiden toteutuksia, vaan ainoastaan (yleensä julkisen) rajapinnan määrittely. Joissain oliokielissä, kuten Javassa, tällaisille puhtaille rajapinnoille on oma syntaksinsa eikä niitä edes varsinaisesti lasketa luokiksi.

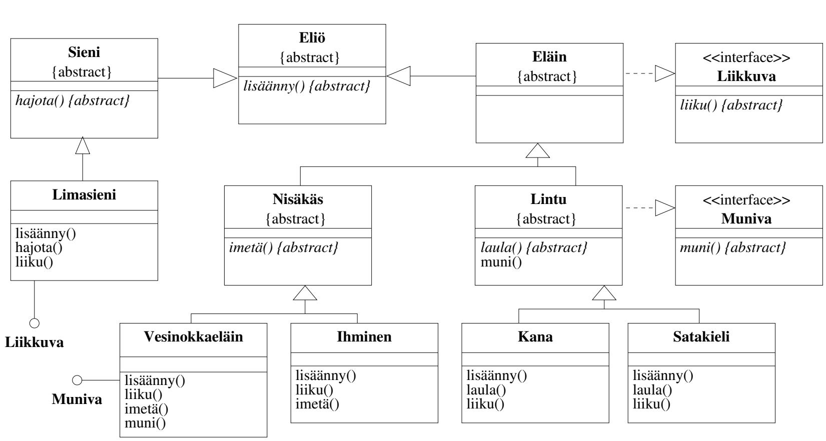
[☺]Tarkasti ottaen C++-standardi joutuu ottamaan kantaa virtuaaliseen periytymiseen purkajienkin yhteydessä, kun se määrittelee purkajien keskinäisen kutsujärjestyksen. Tällä ei kuitenkaan normaalisti ole mitään merkitystä, ja purkajia edelleen kutsutaan käänteisessä järjestyksessä rakentajiin verrattuna.

6.9.1 Rajapintaluokkien käyttö

Rajapintaluokat ovat varsin käteviä, koska niiden avulla voidaan käyttäjälle paljastaa luokkahierarkiasta pelkkä hierarkkinen rajapinta ja kätkeä itse rajapintafunktioiden toteutus konkreettisiin luokkiin. Rajapintaluokkia ja dynaamista sitomista käyttämällä ohjelmoija voi lisäksi itse päättää, millä hierarkiatasolla olioita käsittelee. Joitain funktioita kiinnostaa vain, että niiden parametrit ovat mitä tahansa eläimiä, johonkin tietorakenteeseen talletetaan mitä tahansa sieniä ja niin edelleen.

Usein tulee eteen tilanne, jossa kaikkia haluttuja rajapintoja ei voi millään panna samaan luokkahierarkiaan, koska itse rajapinnat ovat toisistaan riippumattomia ja konkreettisten luokkien rajapinnat ovat erilaisia yhdistelmiä näistä rajapinnoista. Tällaisessa tapauksessa konkreettiset luokat pitäisi pystyä koostamaan erilaisista rajapintakomponenteista luokkahierarkiasta riippumatta. Kuva 6.10 näyttää esimerkin useiden toisistaan riippumattomien rajapintojen käytöstä.

Eri oliokielissä ongelma on ratkaistu eri tavoilla. Ongelmaa ei ole niissä kielissä, joissa ei ole käännösaikaista tyypitystä, koska itse rajapintaluokan käsitettä ei tarvita. Esimerkiksi Smalltalkissa miltä ta-



KUVA 6.10: Luokat, jotka toteuttavat erilaisia rajapintoja

hansa oliolta voidaan pyytää mitä tahansa palvelua ja vasta ohjelman ajoaikana tarkastetaan, pystyykö olio tällaista palvelua tarjoamaan.

Javan erilliset rajapinnat tarjoavat elegantin tavan yhdistellä rajapintoja todellisissa luokissa. Java ei rajoita tällaisten rajapintojen määrää yhteen, vaan luokka voi luetella mielivaltaisen määrän rajapintoja, jotka se toteuttaa. Tällä tavoin luokat voivat luokkahierarkiasta riippumatta toteuttaa erilaisia rajapintoja. Näiden rajapintojen avulla voidaan sitten käsitellä kaikkia rajapinnan toteuttavia luokkia samassa koodissa, koska luokista ei tarvita muuta tietoa kuin se, että ne toteuttavat halutun rajapinnan. Listaus 6.15 näyttää osan kuvan 6.10 luokkien toteutuksesta Javalla.

```
........................... Liikkuva.java ...........................
1 public interface Liikkuva
2 {
3   public void liiku(Sijainti paamaara);
4 }
........................... Muniva.java ...........................
1 public interface Muniva
2 {
3   public void muni();
4 }
........................... Elain.java ...........................
1 public abstract class Elain extends Elio implements Liikkuva
2 {
      ⋮
3 }
........................... Vesinokkaelain.java ...........................
1 public class Vesinokkaelain extends Nisakas implements Muniva
2 {
3   public void lisaanny() { }
4   public void liiku(Sijainti paamaara) { }
5   public void imeta() { }
6   public void muni() { }
      ⋮
7 }
```

LISTAUS 6.15: Erilliset rajapinnat Javassa

6.9.2 C++: Rajapintaluokat ja moniperiytyminen

C++:ssa rajapintaluokkia mallinnetaan abstrakteilla kantaluokilla ja moniperiytymisellä. Mikäli todellinen luokka toteuttaa useita toisistaan riippumattomia rajapintoja, se periytetään kaikista rajapinnat määräävistä abstrakteista kantaluokista. Tällä tavoin saadaan aikaan useita juuriluokkia sisältävä luokkahierarkia, jossa rajapintaluokat ovat kaikkien rajapinnan toteuttavien todellisten luokkien kantaluokkia. Listaus 6.16 näyttää osan kuvan 6.10 toteutuksesta C++:lla.

Mikäli abstraktit kantaluokat sisältävät ainoastaan puhtaita virtu-

```
17 class Liikkuva
18 {
19 public:
20   virtual ~Liikkuva();
21   virtual void liiku(Sijainti paamaara) = 0;
22 };
        ⋮
23 class Muniva
24 {
25 public:
26   virtual ~Muniva();
27   virtual void muni() = 0;
28 };
        ⋮
29 class Elain : public Elio, public Liikkuva
30 {
31 public:
32 private:
33 };
        ⋮
47 class Vesinokkaelain : public Nisakas, public Muniva
48 {
49 public:
50   virtual ~Vesinokkaelain();
51   virtual void lisaanny();
52   virtual void liiku(Sijainti paamaara);
53   virtual void imeta();
54   virtual void muni();
55 };
```

LISTAUS 6.16: Rajapintaluokkien toteutus moniperiytymisellä C++:ssa

aalifunktioita ja ovat näin pelkkiä rajapintaluokkia, ei moniperiytymisen käytöstä aiheudu yleensä ongelmia. Mikäli moniperiytymisessä kantaluokat sen sijaan sisältävät myös rajapintojen toteutuksia ja jäsenmuuttujia, moniperiytyminen aiheuttaa yleensä enemmän ongelmia kuin ratkaisee, kuten aiemmin on todettu. C++:ssa moniperiytymisen järkevä käyttö on jätetty ohjelmoijan vastuulle, eikä kieli itse yritä varjella ohjelmoijaa siinä esiintyviltä vaaroilta.

Rajapintaluokkien toteuttaminen C++:ssa moniperiytymisen avulla aiheuttaa kuitenkin jonkin verran kömpelyyttä ohjelmaan. Ensinnäkin, koska sekä normaali periytyminen että rajapinnan toteuttaminen tehdään kielessä periytymissyntaksilla, ei luokan esittelystä suoraan näe, mitkä sen kantaluokista ovat puhtaita rajapintoja ja mitkä sisältävät myös toteutusta. Tähän ei C++:ssa ole muuta ratkaisukeinoa kuin nimetä rajapintaluokat niin, että käy selvästi ilmi niiden olevan pelkkiä rajapintoja.

Rajapintaluokat, rakentajat ja virtuaalipurkaja

Rajapintaluokat ovat C++:ssa normaaleja luokkia, joten niilläkin on rakentaja, jota kutsutaan aliluokan rakentajasta. Käytännössä tämä rakentaja on kuitenkin aina tyhjä, koska rajapintaluokat nimensä mukaan määrittävät vain rajapinnan eivätkä sisällä jäsenmuuttujia tai toiminnallisuutta. Tämän vuoksi rakentajien kirjoittaminen rajapintaluokille olisi turhauttavaa. Tässä onneksi C++:n normaalisti vaarallinen automatiikka auttaa.

Aliluvussa 3.4.1 todettiin, että jos luokalle ei kirjoiteta yhtään rakentajaa, tekee kääntäjä sinne automaattisesti “tyhjän” oletusrakentajan. Vastaavasti aliluvussa 6.3.2 kävi ilmi, että jos kantaluokan rakentajan kutsu jätetään pois aliluokan rakentajan alustuslistasta, kutsuu kääntäjä automaattisesti kantaluokan oletusrakentajaa. Näiden ominaisuuksien ansiosta rajapintaluokkaan ei tarvitse kirjoittaa rakentajaa ollenkaan, koska tällöin kääntäjä automaattisesti luo tyhjän oletusrakentajan ja kutsuu sitä aliluokissa. Rajapintaluokat ovat C++:ssa lähes ainoa tilanne, jolloin luokalle ei tyyliohjeista poiketen kannata kirjoittaa rakentajaa.

C++:ssa kantaluokkien purkajien tulisi olla virtuaalisia (aliluku 6.5.5), jotta niistä periytettyjen luokkien oliot tuhottaisiin aina oikein. Rajapintaluokat ovat kantaluokkia, joten tämä sääntö koskee myös niitä. Ikävä kyllä, jos rajapintaluokan esittelyssä esitellään

purkaja virtuaaliseksi, pitää tälle purkajalle kirjoittaa myös toteutus, vaikka kyseessä onkin pelkkä rajapintaluokka ilman toiminnallisuutta tai dataa, ja näin purkajan toteutus on aina tyhjä.

Ärsyttäväksi tämän vaatimuksen virtuaalipurkajan toteutuksesta tekee se, että koska rajapintaluokassa ei muuten ole toiminnallisuutta, olisi luontevaa kirjoittaa sille pelkkä otsikkotiedosto (`.hh`) ja jättää varsinainen kooditiedosto (`.cc`) kokonaan kirjoittamatta. Purkajan toteutus taas normaalisti kirjoitettaisiin juuri kooditiedostoon. Tähän ongelmaan löytyy onneksi ratkaisu. C++ antaa mahdollisuuden kirjoittaa jäsenfunktion toteutuksen suoraan luokkaesittelyn *sisälle* (samaan tapaan kuin Javassa tehdään). Tämä ei ole normaalisti hyvää tyyliä, koska luokan toteutusta ja esittelyä ei ole syytä sotkea keskenään. Lisäksi tällaiset luokan esittelyyn upotetut jäsenfunktiot optimoidaan aina kuten **`inline`**-funktiot (liitteen aliluku A.2), mikä ei yleisessä tapauksessa ole välttämättä hyvä asia.

Koska kuitenkin rajapintaluokan purkaja on aina tyhjä, voi tästä tyylisäännöstä ainakin tämän kirjan kirjoittajien mielestä poiketa tässä tilanteessa. Näin rajapintaluokan purkaja saadaan kirjoitettua kokonaan otsikkotiedostoon eikä erillistä toteutustiedostoa tarvita. Listaus 6.17 näyttää esimerkkinä rajapintaluokan `Liikkuva` esittelyn näin kirjoitettuna.

6.9.3 Ongelmia rajapintaluokkien käytössä

Erilliset rajapinnat tai rajapintaluokat selkeyttävät luokkahierarkiaa ja mahdollistavat kätevästi sen, että saman rajapinnan toteuttavia olioita voidaan käsitellä rajapintaosoittimen tai -viitteen avulla riippumatta luokkien sijainnista luokkahierarkiassa. Rajapintaluokat eivät kuitenkaan ratkaise kaikkia ongelmia.

```
1 class Liikkuva
2 {
3 public:
4    // Kääntäjä tuottaa automaattisesti tyhjän oletusrakentajan
5    virtual ~Liikkuva() {} // Upotettu tyhjä virtuaalipurkaja (inline)
6    virtual void liiku(Sijainti paamaara) = 0;
7 };
```

LISTAUS 6.17: Rajapintaluokan purkaja esittelyyn upotettuna

Rajapintojen nimien päällekkäisyys

Moniperiytymisen yhteydessä todettiin, että monen kantaluokan tapauksessa voi käydä niin, että usean kantaluokan rajapinnassa on täsmälleen samanniminen jäsenfunktio samoilla parametreilla. Täsmälleen sama ongelma voi ilmetä myös rajapintaluokkien tapauksessa, joten ongelma koskee myös esimerkiksi Javaa, vaikka siinä ei varsinaista moniperiytymistä olekaan.

Samannimisten jäsenfunktioiden erottaminen toisistaan käy C++:ssa aliluvussa 6.8.1 sivulla 184 esitetyllä tekniikalla apukantaluokkia käyttämällä. Tällä tavoin rajapinnat toteuttava luokka voi antaa eri toteutuksen kumpaakin rajapintaa varten. Tämä tekniikka vaatii kielessä kuitenkin aitoa moniperiytymistä, joten sitä ei, ikävä kyllä, voi käyttää Javassa. Tämän kirjan kirjoittajat eivät valitettavasti onnistuneet löytämään tai keksimään yhtään tapaa, jolla ongelman voisi ratkaista Javan keinoin.

Rajapintaluokkien yhdistäminen

Rajapintaluokat voivat aiheuttaa myös ongelmia itse periytymisessä. Oletetaan esimerkiksi, että halutaan kirjoittaa funktio `menePesaanJaMuni`, joka käskee eliötä ensin liikkumaan pesäänsä ja sen jälkeen munimaan sinne. Olisi luonnollista, että tällaiselle funktiolle voisi antaa parametrina minkä tahansa olion, joka pystyy *sekä* liikkumaan *että* munimaan. `Liikkuva` ja `Muniva` ovat kuitenkin erillisiä rajapintaluokkia, eikä funktion parametrilla voi olla kuin yksi tyyppi.

Yksi ratkaisuyritys olisi luoda uusi rajapintaluokka `LiikkuvaJaMuniva`, johon periyttämällä yhdistetään molemmat tarvittavat rajapinnat.[Ω]

```
class LiikkuvaJaMuniva : public Liikkuva, public Muniva
{
};
```

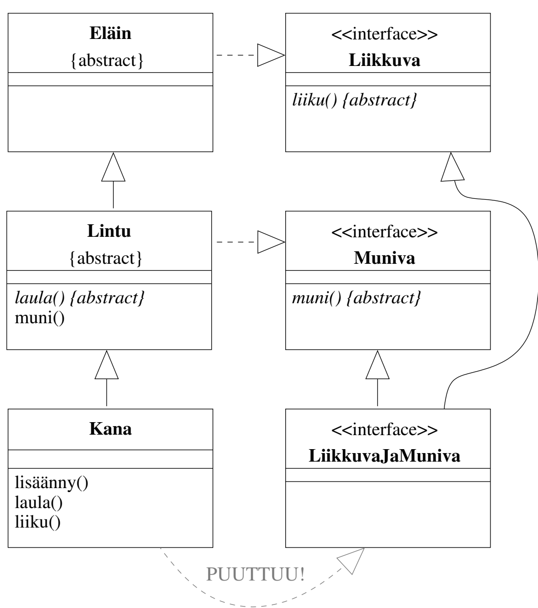
[Ω]Rajapintaluokan `LiikkuvaJaMuniva` purkaja on automaattisesti virtuaalinen, koska tämä ominaisuus periytyy sen kantaluokilta. Tämän vuoksi virtuaalipurkajan esitteleminen ei ole välttämätöntä, vaikka ehkä kylläkin hyvää tyyliä.

Tämän uuden rajapintaluokan avulla funktio olisi helppo esitellä seuraavasti:

```
void menePesaanJaMuni(LiikkuvaJaMuniva& emo);
```

Vaikka edellä esitetty tapa tuntuukin järkevältä, se ei yllättäen toimi. Jos funktiolle yrittää esimerkiksi antaa parametrina luokan `Kana` oliota, valittaa kääntäjä ettei `Kana`-olio kelpaa funktion parametriksi. Syynä tähän on, että vaikka `Kana` toteuttaakin *erikseen* periytymisen kautta sekä rajapinnan `Liikkuva` että `Muniva`, se ei luokkahierarkian mukaan kuitenkaan ole periytetty näiden yhdistelmästä `LiikkuvaJaMuniva`. Niinpä `Kana`-olio ei kuulu tähän luokkaan eikä kelpaa parametriksi. Kuva 6.11 havainnollistaa tilannetta.

Kaikki tämä seuraa suoraan periytymisen ominaisuuksista. Vaikka luokkaan `LiikkuvaJaMuniva` ei olekaan lisätty yhtään ylimääräistä jäsenfunktiota luokkiin `Liikkuva` ja `Muniva` verrattuna, saat-



KUVA 6.11: Ongelma yhdistettyjen rajapintojen kanssa

taa olla että tämän luokan *semantiikka* (merkitys) sisältää muutakin kuin kahden rajapinnan yhdistämisen. Esimerkiksi rajapintaluokka `LentavaJaMuniva` näyttäisi määrittelyltään samalta, mutta siinä voisi olla dokumentoituna, että liikkuminen tapahtuu lentäen. Tällöin `Kanan` ei tietysti pitäisikään kelvata toteuttamaan tätä rajapintaa.

Ainoa ratkaisu tähän ongelmaan C++:n ja Javan tapaisissa kielissä olisi luoda ohjelman luokkahierarkiaan varsinaisten rajapintaluokkien lisäksi myös kaikki näiden yhdistelmät ja merkitä erikseen hierarkiaan, mitkä luokat toteuttavat myös nämä yhdistelmät. Tämä kuitenkin tekisi luokkahierarkioista erittäin sotkuisia, ja yhdistelmien määrä räjähtäisi helposti todella suureksi. Näissä kielissä ei olekaan mitään eleganttia ratkaisua tähän ongelmaan. Smalltalkissa puolestaan tätä ongelmaa ei tule, koska siinä ei ole tarvetta rajapintaluokille laisinkaan. Toisaalta tämä taas johtuu siitä, että koko kieli ei tunne rajapinnan käsitettä, koska kaikki tyyppitarkastukset tehdään kielessä vasta ajoaikana.

6.10 Periytyminen vai kooste?

Milloin luokkien välillä on olemassa periytymissuhde ja milloin on kyse koosteesta? Koosteessa on kyse kahden tai useamman *olion* muodostamasta kokonaisuudesta ja periytymisessä taas yhdestä oliosta, jonka luokkarakenne koostuu usean luokan ominaisuuksista. Vaikka kyse on kahdesta hyvin erilaisesta asiasta, niin suunnittelussa on usein vaikeata tehdä valintaa koosteen ja periytymisen välillä.

Perinteinen esimerkki oliosuunnittelussa on ihmisten (kohtuuton) yksinkertaistaminen mallintamalla heidän ominaisuuksiaan erillisinä luokkina: insinööri, isä, viulisti ja niin edelleen. (Ei attribuutteina, sillä nämä ominaisuudet itsessään ovat mutkikkaita, ja siten luokan arvoisia.) Halutessamme mallin Sibelius-akatemiasta ja TTY:lta valmistuneesta ohjelmistosuunnittelijasta voimme katsoa hänellä olevan sekä Insinöörin ja Viulistin ominaisuudet, joten periytyminen näistä luokista olisi perusteltua. Mutta mitäpä silloin kun hän muutaman vuoden kuluttua saa lapsia ja palkanmaksujärjestelmämme pitäisi liittää häneen myös isää kuvaavan luokan ominaisuudet? Nyt joudummekin tekemään häntä kuvaamaan kokonaan uuden olion, jonka rakenne on periytetty luokista Insinööri, Viulisti ja Isä. Mitä enemmän erilaisia vaihtoehtoja mallinnamme, sitä enemmän jou-

dumme tekemään eri tavoin periytettyjä luokkia ja ohjelmassa vaihtamaan ihmisiä kuvaavia olioita heidän elämäntilanteidensa muuttuessa. Selvästi parempi vaihtoehto tässä dynaamisessa tilanteessa on liittää tarvittava määrä ihmiseen liittyviä "ominaisuuksia" koosteella. Kuva 6.12 esittää molemmat mahdollisuudet.

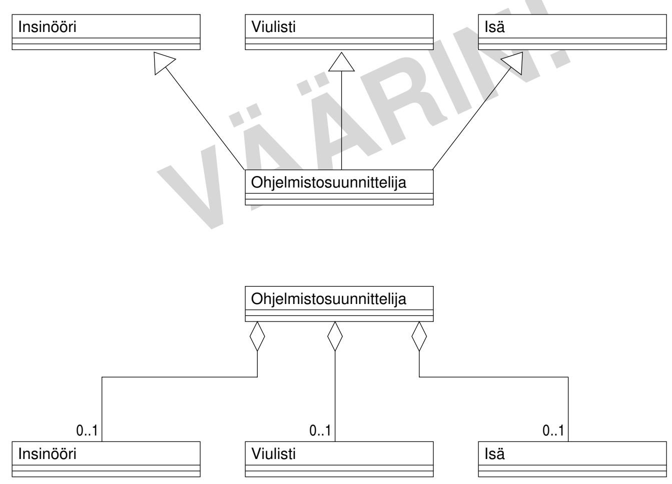
Periytymisen "nyrkkisäännöiksi" suositellaan: [Meyer, 1997]

1. Periytä luokka B luokasta A vain silloin, kun pystyt perustelemaan, että luokan B oliot voidaan ***aina*** nähdä myös luokan A olioina.

2. Älä käytä periytymistä, jos periytymisellä saatu olion rakenne voi muuttua ohjelman ajoaikana (käytä tällöin koostetta).

6.11 Sovelluskehykset

Ohjelmistojen toteutuksessa voidaan hyödyntää aiemmissa projekteissa tehtyjä tai projektin ulkopuolelta ostettuja ohjelmakomponent-

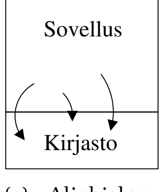


KUVA 6.12: Insinööri, viulisti ja isä

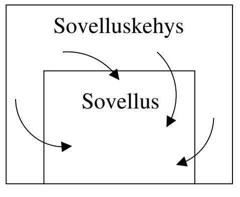
teja. Nämä komponentit ovat usein luokkakirjastoja, joiden avulla saadaan käyttöön valmiita olioita. Ohjelmiston toiminnallinen logiikka eli olioiden välinen yhteistyö (kommunikointi) on edelleen täysin toteuttajan vastuulla ja hallinnassa.

Ohjelmistoa voidaan ryhtyä rakentamaan myös siten, että valmis komponentti määrittelee etukäteen myös toiminnallisuuden kannalta olioiden kommunikointiin liittyviä vaatimuksia. Tällaista rakennetta sanotaan **sovelluskehykseksi** (*framework*). Olio-ohjelmoinnissa sovelluskehykset ovat luokkakirjastoja, joista on tarkoitus periyttämällä erikoistaa oman sovelluksen toteuttavat luokkien versiot. Kuvassa 6.13 näkyy ero sovelluskehysohjelman rakenteessa verrattuna “perinteiseen” aliohjelmakirjastorakenteeseen. Kirjastorutiineja kutsutaan toteutetusta sovelluksesta, jolla on vastuu kokonaiskontrollista. Sovelluskehyksessä kehys on määritellyt perusrakenteen, jonka osaset kutsuvat kehyksen määrittelemissä tapahtumissa sovelluksen operaatioita (jotka on toteutettu esimerkiksi kehyksen koodista periytetyissä luokissa).

Sovelluskehys tavallaan määrittelee koko sovelluksen “tunnelman” tai arkkitehtuurin, jonka puitteissa ohjelmoijan on toimittava. Ikkunointijärjestelmän sovelluskehys voi määritellä tarkasti, mitä piirto-operaatioita ja tapahtumatietoja ohjelmoijan on toteutettava. Järjestelmä voi tarjota vaikkapa “perusikkunan”, jonka luokasta periyttämällä toteutetaan varsinainen oma sovellus.



(a) Aliohjelmakirjasto



(b) Sovelluskehys

KUVA 6.13: Aliohjelmakirjasto ja sovelluskehys

Luku 7

Lisää olioiden elinkaaresta

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

– CATCH-22 *[Heller, 1961]*

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

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

7.1 Olioiden kopiointi

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

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

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

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

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

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

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

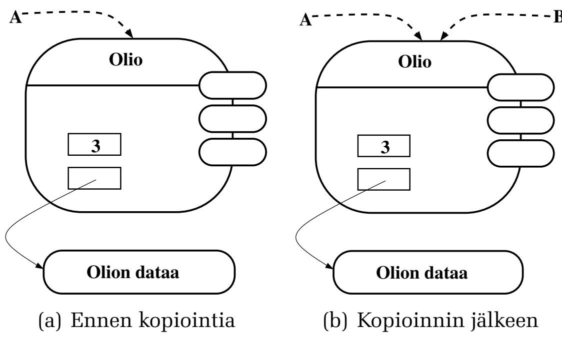
7.1.1 Erilaiset kopiointitavat

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

Viitekopiointi

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

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



(a) Ennen kopiointia (b) Kopioinnin jälkeen

KUVA 7.1: Viitekopiointi

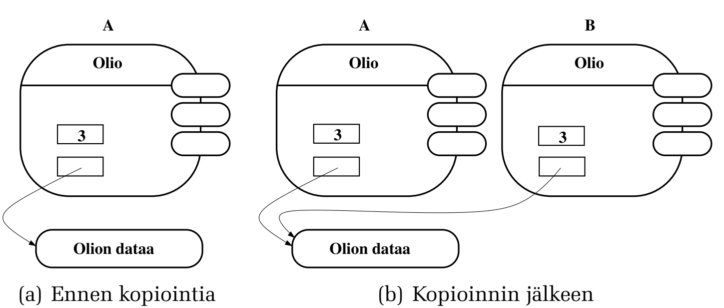
naamisesti. Tällaisia kieliä ovat esimerkiksi Java ja Smalltalk. Kun näissä kielissä luodaan uusi muuttuja ja alustetaan se olemassa olevalla muuttujalla, viittaavat molemmat muuttujat tämän jälkeen *samaan* olioon. C++:ssa sen sijaan viitekopiointia käytetään vain, kun erikseen luodaan viitteitä olioiden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

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

Matalakopiointi

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

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



(a) Ennen kopiointia (b) Kopioinnin jälkeen

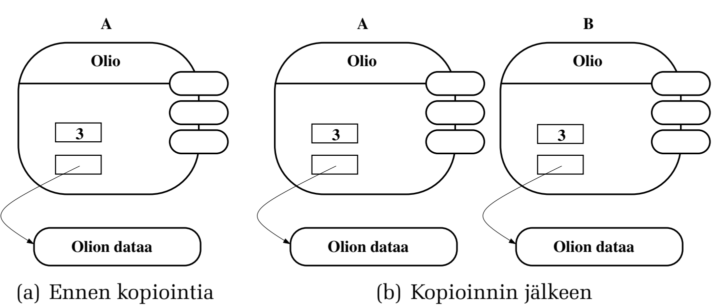
KUVA 7.2: Matalakopiointi

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

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

Syväkopiointi

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



(a) Ennen kopiointia (b) Kopioinnin jälkeen

KUVA 7.3: Syväkopiointi

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

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

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

7.1.2 C++: Kopiorakentaja

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

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

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

Periytyminen ja kopiorakentaja

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

```
.................................. mjono.hh ..................................
 1 class Mjono
 2 {
 3 public:
 4    Mjono(char const* merkit);
 5    Mjono(Mjono const& vanha); // Kopiorakentaja
 6    virtual ~Mjono();
          ⋮
11 private:
12    unsigned long koko_;
13    char* merkit_;
14 };
.................................. mjono.cc ..................................
 1 Mjono::Mjono(Mjono const& vanha) : koko_(vanha.koko_), merkit_(0)
 2 {
 3    if (koko_ != 0)
 4    { // Varaa tilaa, jos koko ei ole nolla
 5       merkit_ = new char[koko_ + 1];
 6       for (unsigned long i = 0; i != koko_; ++i)
 7          { merkit_[i] = vanha.merkit_[i]; } // Kopioi merkit
 8       merkit_[koko_] = '\0'; // Loppumerkki
 9    }
10 }
```

LISTAUS 7.1: Esimerkki kopiorakentajasta

pmjono.hh
```
1  class PaivattyMjono : public Mjono
2  {
3  public:
4    PaivattyMjono(char const* merkit, Paivays const& paivays);
5    PaivattyMjono(PaivattyMjono const& vanha); // Kopiorakentaja
6
7    virtual ~PaivattyMjono();
       ⋮
15 private:
16   Paivays paivays_;
17 };
```
pmjono.cc
```
1  // Olettaa, että Paivays-luokalla on kopiorakentaja
2  PaivattyMjono::PaivattyMjono(PaivattyMjono const& vanha)
3    : Mjono(vanha), paivays_(vanha.paivays_)
4  {
5  }
```
LISTAUS 7.2: Kopiorakentaja aliluokassa

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

Kääntäjän luoma oletusarvoinen kopiorakentaja

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

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

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

Kopioinnin estäminen

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

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

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

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

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

7.1.3 Kopiointi ja viipaloituminen

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

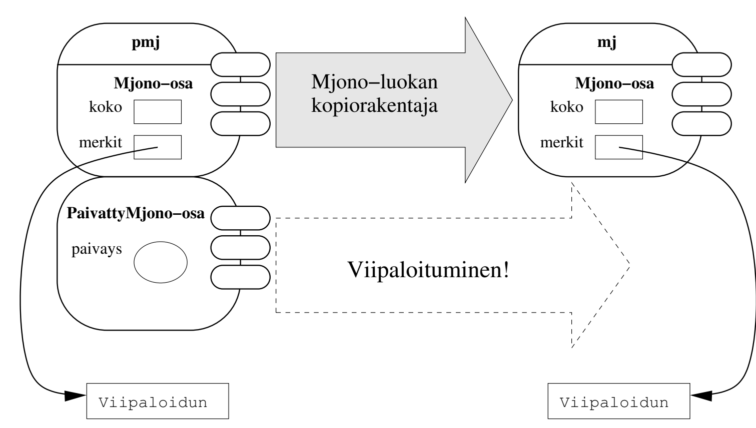
Kun C++:ssa luodaan kopiorakentajalla oliosta kopio, täytyy kopiota luovan ohjelmanosan tietää, minkä luokan oliota ollaan luomassa. Saman voi sanoa myös niin, että C++:ssa ei ole suoraan mitään tapaa luoda oliota ilman, että käännösaikana tiedetään sen tyyppi (tämä koskee sekä kopiointia että muutakin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

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

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

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

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



KUVA 7.4: Viipaloituminen olion kopioinnissa

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

Viipaloitumisen esiintyminen

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

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

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

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

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

Viipaloituminen ja muut oliokielet

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

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

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

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

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

Viipaloitumisen kiertäminen C++:ssa

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

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

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

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

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

LISTAUS 7.3: Viipaloitumisen kiertäminen `kloonaa`-jäsenfunktiolla

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

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

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

7.2 Olioiden sijoittaminen

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

7.2.1 Sijoituksen ja kopioinnin erot

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

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

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

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

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

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

7.2.2 C++: Sijoitusoperaattori

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

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

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

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

mjono.hh

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

mjono.cc

```
1  Mjono& Mjono::operator =(Mjono const& vanha)
2  {
3    if (this != &vanha)
4    { // Jos ei sijoiteta itseen
5      delete[] merkit_; merkit_ = 0; // Vapauta vanha
6      koko_ = vanha.koko_; // Sijoita koko
7      if (koko_ != 0)
8      { // Varaa tilaa, jos koko ei nolla
9        merkit_ = new char[koko_ + 1];
10       for (unsigned long i = 0; i != koko_; ++i)
11       { // Kopioi merkit
12         merkit_[i] = vanha.merkit_[i];
13       }
14       merkit_[koko_] = '\0'; // Loppumerkki
15     }
16   }
17   return *this;
18 }
```

LISTAUS 7.4: Esimerkki sijoitusoperaattorista

jusijoituksesti aiheutuu lopulta jäsenfunktioketju

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

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

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

Sijoitus itseen

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

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

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

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

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

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

Periytyminen ja sijoitusoperaattori

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

Kääntäjän luoma oletusarvoinen sijoitusoperaattori

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

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

```
................................ pmjono.hh ................................
 1 class PaivattyMjono : public Mjono
 2 {
 3 public:
14   PaivattyMjono& operator =(PaivattyMjono const& vanha);
       ⋮
17 };
................................ pmjono.cc ................................
 1 PaivattyMjono& PaivattyMjono::operator =(PaivattyMjono const& vanha)
 2 {
 3   if (this != &vanha)
 4   { // Jos ei sijoiteta itseen
 5     Mjono::operator =(vanha); // Kantaluokan sijoitusoperaattori
 6     // Oma sijoitus, oletetaan että Paivays-luokalla on sijoitusoperaattori
 7     paivays_ = vanha.paivays_;
 8   }
 9   return *this;
10 }
```

LISTAUS 7.5: Sijoitusoperaattori periytetyssä luokassa

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

Sijoituksen estäminen

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

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

7.2.3 Sijoitus ja viipaloituminen

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

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

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

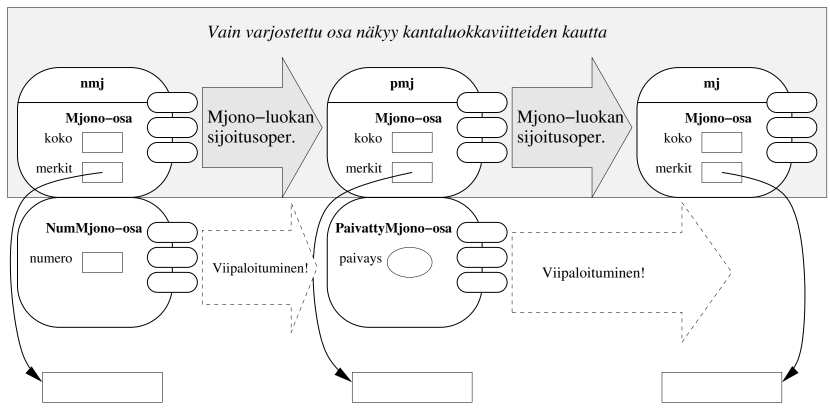
Viipaloitumisvaaran estämiskeinoja C++:ssa on muutama. Helpoin on tehdä luokkahierarkia, jossa kaikki kantaluokat ovat abstrakteja, jolloin niillä ei ole sijoitusoperaattoria ollenkaan. Tällöin viipaloitumista ei voi tapahtua [Meyers, 1996, Item 33]. Toinen ratkaisu on tarkastaa aina sijoituksen yhteydessä,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

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

LISTAUS 7.6: Viipaloitumismahdollisuudet sijoituksessa

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

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



KUVA 7.5: Viipaloituminen olioiden sijoituksessa

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

LISTAUS 7.7: Viipaloitumisen estäminen ajoaikaisella tarkastuksella

7.3 Oliot arvoparametreina ja paluuarvoina

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

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

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

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

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

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

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

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

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

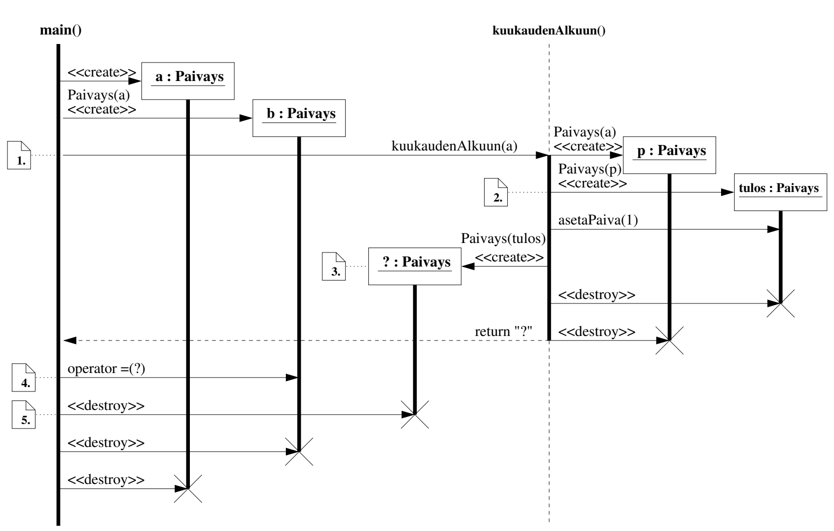
LISTAUS 7.8: Olio arvoparametrina ja paluuarvona

aivan yhtä itsestäänselvää vaihtoehtoa kuin viitteet parametrinvälityksen tapauksessa.

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

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

1. Funktiokutsun yhteydessä parametrista a tehdään automaatti-



KUVA 7.6: Oliot arvoparametreina ja paluuarvoina

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

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

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

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

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

7.4 Tyyppimuunnokset

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

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

kuin numeerinen. Jos osoitin muunnetaan kokonaisluvuksi, tuloksena on useimmissa ympäristöissä kokonaisluku, joka ilmoittaa sen koneen muistiosoitteen, jossa osoittimen osoittama olio sijaitsee.

Olio-ohjelmoinnissa on tavallista, että ohjelmoija kirjoittaa omia abstrakteja tietotyyppejään, kuten murtolukuja, kompleksilukuja ja niin edelleen. Tämän vuoksi C++:ssa on mahdollisuus kirjoittaa omia tyyppimuunnosoperaattoreita, joiden avulla on mahdollisuus tehdä tyyppimuunnoksia omien tietotyyppien ja muiden tyyppien välillä. Tämän lisäksi standardoinnin yhteydessä C++:aan tuli tyyppimuunnoksille uusi syntaksi, jota nykyisin suositellaan käytettäväksi vanhan C:stä periytyvän muunnossyntaksin sijaan.

7.4.1 C++:n tyyppimuunnosoperaattorit

C-kielessä tyyppimuunnokset tehtiin syntaksilla `(uusiTyyppi)vanhaArvo`. Tämän syntaksin ongelma on, että sulut ovat hieman epäloogisessa paikassa — eivät parametrin vaan itse operaation ympärillä. Tästä johtuen C++-kieleen lisättiin vaihtoehtoinen syntaksi `uusiTyyppi(vanhaArvo)`.

Kummankin näiden tyyppimuunnoksen ongelma on, että niitä voidaan käyttää suorittamaan kaikentyyppisiä muunnoksia — niin muunnoksia kokonaisluvusta liukuluvuksi kuin muunnoksia olio-osoittimesta kokonaisluvuksi. Kääntäjä ei tarkasta lähes ollenkaan muunnoksen mielekkyyttä vaan luottaa siihen, että ohjelmoija tietää mitä on tekemässä. Käytännössä tämä on johtanut siihen, että tyyppimuunnoksiin jää helposti kirjoitusvirheitä. Tämän lisäksi tyyppimuunnoksista johtuvien virheiden löytäminen on usein vaikeaa, koska itse muunnokset näyttävät aivan funktiokutsuilta, joten niiden löytäminen koodista ei ole aivan helppoa.

ISOC++ on pyrkinyt korjaamaan tätä ongelmaa lisäämällä kieleen neljä aivan uutta tyyppimuunnosoperaattoria, joiden syntaksi poikkeaa jonkin verran aiemmista:

```
static_cast<uusiTyyppi>(vanhaArvo)
const_cast<uusiTyyppi>(vanhaArvo)
dynamic_cast<uusiTyyppi>(vanhaArvo)
reinterpret_cast<uusiTyyppi>(vanhaArvo)
```

Syntaksi saattaa ensikatsomalta tuntua oudolta, mutta se on yhteensopiva mallien käyttämän syntaksin kanssa (malleista kerrotaan luvussa 9.5).

Jokainen näistä operaattoreista on tarkoitettu vain tietynlaisten mielekkäiden muunnosten tekemiseen, ja kääntäjä antaa virheilmoituksen, jos niitä yritetään käyttää väärin. Myös vanhat tyyppimuunnossyntaksit on säilytetty kielessä yhteensopivuussyistä, mutta niiden käyttöä tulisi välttää, ja suosia uusia operaattoreita.

static_cast

Tyyppimuunnoksen **static_cast** tehtävänä on suorittaa kaikkia sellaisia tyyppimuunnoksia, joiden mielekkyydestä kääntäjä voi varmistua jo käännösaikana. Tämä tarkoittaa esimerkiksi muunnoksia eri kokonaislukutyyppien välillä, muunnoksia **enum**-luettelotyypeistä kokonaisluvuiksi ja takaisin sekä muunnoksia kokonaislukutyyppien ja liukulukutyyppien välillä. Esimerkiksi kahden kokonaisluvun keskiarvon voi laskea liukulukuna seuraavasti:

```
double ka =
(static_cast<double>(i1) + static_cast<double>(i2)) / 2.0;
```

Muunnos **static_cast** ei suostu suorittamaan sellaisia muunnoksia, jotka eivät ole mielekkäitä. Esimerkiksi muunnos

```
Paivays* pvmp = new Paivays();
int* ip = static_cast<int*>(pvmp); // KÄÄNNÖSVIRHE!
```

antaa käännösaikana virheilmoituksen, koska päiväysosoittimen päässä ei voi olla kokonaislukua ja näin ollen muunnos päiväysosoittimesta kokonaislukuosoittimeksi on järjetön.

Tavallisten tyyppimuunnosten lisäksi **static_cast** on mahdollinen myös silloin, kun kantaluokkaosoittimen päässä *tiedetään* olevan jonkin aliluokan olio, johon halutaan aliluokkaosoitin *ilman, että olion tyyppiä testataan ajoaikana*. Aiemmin aliluvussa 6.5.3 käsitelty **dynamic_cast** on normaalisti oikea väline tähän, mutta jos kantaluokkaosoittimen päässä olevan olion tyyppi on todella tiedossa, voi nopeuskriittisissä ohjelmissa olla tarve päästä eroon olion tyypin testauksesta. Tällöin muunnoksen voi korvata nopeammalla mutta turvattomammalla **static_cast**-operaatiolla.

const_cast

Normaalisti **const**-sanan käyttö ohjelmissa lisää ohjelman luotettavuutta, koska se pakottaa ohjelmoijan miettimään, mitä olioita muutetaan ja mitä ei. Vastaavasti se estää sellaisten olioiden muuttamisen, jotka on aiemmin määritelty vakioiksi. Joskus on kuitenkin pakko käyttää sellaista ohjelmakoodia, joka on suunniteltu "väärin" ja jossa **const**-sanan käyttö tuo ongelmia. Tällöin **const_cast** antaa ohjelmoijalle mahdollisuuden poistaa **const**-sanan vaikutuksen, eli sillä voi tehdä vakio-osoittimesta ja -viitteestä ei-vakio-osoittimen tai -viitteen.

Yksi tyypillinen esimerkki näkyy listauksessa 7.9. Listauksen ohjelmassa käytetään jonkin toisen tahon ohjelmoimaa kirjastoa kirjojen käsittelyyn. Tästä kirjastosta löytyy funktio `kauankoPalautukseen`, joka laskee annetusta kirjasta, kuinka pitkä aika viimeiseen palautuspäivään on. Ongelmana on, että tämän funktion tekijä ei ole kuullutkaan **const**-sanasta, joten funktio ottaa parametrinaan normaaliviitteen kirjaan vakioviitteen sijaan.

Funktio `tulostaPalautusaika` sen sijaan on koodattu oikein ja ottaa parametrinaan vakioviitteen. Jos funktiossa nyt yritettäisiin kutsua virheellistä funktiota, kääntäjä ei hyväksyisi tätä kutsua, koska vakio-oliota ei voi laittaa ei-vakioviitteen päähän. On kuitenkin selvää, että kyseinen virhe ei johdu varsinaisesti virheellisestä ohjelmasuunnittelusta vaan siitä, että toinen ohjelmoija on unohtanut

```
1  int kauankoPalautukseen(KirjastonKirja& kirja)
2  {
3    // Funktion toteutus ei muuta kirja-oliota mitenkään
       ⋮
14 }
```

```
1  void tulostaPalautusaika(KirjastonKirja const& k)
2  {
3    //   int paivia = kauankoPalautukseen(k) aiheuttaisi käännösvirheen!
4    int paivia = kauankoPalautukseen(const_cast<KirjastonKirja&>(k));
5    cout << "Kirjan " << k.annaNimi() << " palautukseen on ";
6    cout << paivia << " päivää." << endl;
7  }
```

LISTAUS 7.9: Esimerkki **const_cast**-muunnoksesta

const-sanan. Tämän vuoksi rivillä 4 on käytetty **const_cast**-muunnosta tuottamaan ei-vakioviite, joka kelpaa parametriksi huolimattomasti kirjoitetulle funktiolle.

Koska **const_cast**-muunnoksen käyttö rikkoo C++:n "vakiota ei voi muuttaa" -periaatetta vastaan, sen käyttö on periaatteessa aina osoitus siitä, että jokin osa ohjelmasta on suunniteltu huonosti. Tästä syystä sen käyttöä tulisi välttää aina kun mahdollista ja mieluummin korjata se ohjelmanosa, jossa **const**-sanan käyttö on laiminlyöty.

dynamic_cast

Tyyppimuunnosta **dynamic_cast** käytetään muuttamaan kantaluokkaosoitin tai -viite aliluokkaosoittimeksi tai -viitteeksi ja samalla tarkastamaan, että osoittimen tai viitteen päässä oleva olio on todella sopivaan luokkaan kuuluva. Sen toiminta on jo käsitelty aliluvussa 6.5.3 sivulla 164.

Ajoaikaisesta tarkastuksesta johtuen **dynamic_cast** on ainoa ISOC++:n uusista tyyppimuunnoksista, jota ei voi mitenkään toteuttaa vanhaa tyyppimuunnossyntaksia käyttäen.

reinterpret_cast

Kaikki tähän saakka esitellyt tyyppimuunnosoperaattorit ovat suostuneet tekemään vain sellaisia muunnoksia, jotka ovat jollain tavalla käytettyjen tyyppien kannalta mielekkäitä (ellei **const_cast**-muunnoksen tekemää vakioisuuden poistoa katsota "mielettömäksi"). Silloin tällöin ohjelmissa saattaa kuitenkin joutua käsittelemään tietoa tavalla, joka ei ole sen todellisen tyypin mukainen.

Esimerkiksi osoitinta voi joskus joutua käsittelemään muistiosoitteena — siis kokonaislukuna. Vastaavasti monissa käyttöliittymäkirjastoissa painonappeihin voi liittää "tunnistustietoa", joka välitetään ohjelmalle nappia painettaessa. Tämä tieto voi olla vaikkapa tyyppiä **void***, mutta silti siihen pitäisi saada talletettua (ja myöhemmin palautettua) osoitin johonkin olioon.

Tällaisia tiedon esitystavan muuttamiseen liittyviä muunnoksia varten on operaattori **reinterpret_cast**. Kuten sen nimikin ilmaisee, se "tulkitsee uudelleen" sille annetun tiedon toisen tyyppisenä. Muunnoksen lähes ainoa käyttökohde on muuntaa tieto ensin toisen-

tyyppiseksi ja myöhemmin takaisin. ISOC++ -standardi luettelee seuraavat muunnostyypit, joihin **reinterpret_cast** kelpaa:

- Osoittimen voi muuntaa kokonaisluvuksi, jos kokonaislukutyyppi on niin suuri, että osoitin mahtuu siihen.
- Vastaavasti kokonaisluvun (tai luettelotyypin arvon) voi muuntaa takaisin osoittimeksi.
- Tavallisen osoittimen voi muuntaa toisentyyppiseksi osoittimeksi.
- Viitteen voi muuntaa toisentyyppiseksi viitteeksi.
- Funktio-osoittimen voi muuntaa toisentyyppiseksi funktio-osoittimeksi (mutta tavallista osoitinta ei tarkasti ottaen voi muuntaa funktio-osoittimeksi tai päinvastoin, ainakaan niin että koodi olisi siirrettävää). Sama pätee jäsenfunktio- ja -muuttujaosoittimille.

Kaikissa näissä muunnoksissa muunnettua arvoa ei pitäisi käyttää mihinkään muuhun kuin tiedon tallettamiseen ja myöhemmin takaisin muuntamiseen. Ainoana poikkeuksena on osoittimen muuntaminen kokonaisluvuksi, jonka tuottaman tuloksen pitäisi olla "sellainen, että se ei hämmästytä niitä, jotka tuntevat kyseisen koneen muistiosoitteiden rakenteen".[♊]

Listauksessa 7.10 seuraavalla sivulla on esimerkki tilanteesta, jossa käyttöliittymäkirjasto tarjoaa funktion `luoNappula`, jolle annetaan nappulan nimi ja siihen liittyvä kokonaisluku. Kun nappulaa painetaan, käyttöliittymä kutsuu funktiota `nappulaaPainettu` ja antaa sille parametriksi nappulaan liittyvän kokonaisluvun. Esimerkki käyttää riveillä 4–5 **reinterpret_cast**-muunnosta tallettamaan nappuloiden sisältämiin lukuihin osoittimia kirjoihin ja myöhemmin rivillä 10 muuntamaan saadut kokonaisluvut taas takaisin osoittimiksi.

7.4.2 Ohjelmoijan määrittelemät tyyppimuunnokset

Kun ohjelmoija kirjoittaa oman tietotyyppinsä, on mahdollisesti tarvittavia tyyppimuunnoksia kahdenlaisia: tyyppimuunnoksia, jotka

[♊] "It [the mapping function] is intended to be unsurprising to those who know the addressing structure of the underlying machine." [ISO, 1998, kohta 5.2.10.4]

```
⋮
1  void luoNappula(string const& nimi, unsigned long int luku);
⋮
```

```
1  // Tämä funktio luo käyttöliittymän
2  void luoKayttoliittyma(KirjastonKirja* kirja1, KirjastonKirja* kirja2)
3  {
4    luoNappula("Kirja1", reinterpret_cast<unsigned long int>(kirja1));
5    luoNappula("Kirja2", reinterpret_cast<unsigned long int>(kirja2));
6  }
⋮
7  // Tätä funktiota kutsutaan, kun nappulaa painetaan
8  void nappulaaPainettu(unsigned long int luku)
9  {
10   KirjastonKirja* kp = reinterpret_cast<KirjastonKirja*>(luku);
11   cout << "Painettu kirjaa " << kp->annaNimi() << "." << endl;
12 }
```

LISTAUS 7.10: Tiedon esitystavan muuttaminen ja `reinterpret_cast`

muuntavat muun tyyppisiä arvoja ohjelmoijan kirjoittamaksi tyypiksi, ja muunnoksia, jotka muuttavat ohjelmoijan oman tyypin arvoja muuntyyppisiksi. C++ antaa ohjelmoijalle mahdollisuuden kirjoittaa omia tyyppimuunnoksia molempiin suuntiin. Tyyppimuunnokset kirjoitetaan oman tietotyyppiluokan jäsenfunktioina, ja kumpaankin suuntaan tapahtuvilla muunnoksilla on oma syntaksinsa.

Rakentaja tyyppimuunnoksena

Oleellisilta osiltaan tyyppimuunnos on uuden erityyppisen arvon luomista olemassa olevan arvon pohjalta. C++:ssa uusia "arvoja" (olioita) luodaan rakentajan avulla, joten on luontevaa yhdistää tyyppimuunnokset ja rakentajat. Tämä tapahtuu niin, että kielen mukaan *jokainen yksiparametrinen rakentaja* on myös tyyppimuunnos rakentajan parametrin tyypistä luokan olioksi.

Esimerkiksi listauksessa 7.11 seuraavalla sivulla on osa luokan `Murtoluku` määrittelyä. Luokalla on rakentaja, jonka avulla uuden murtoluvun voi luoda kokonaisluvun avulla. Tällöin luodaan uusi murtoluku, joka on arvoltaan sama kuin parametrina oleva kokonais-

```
1  class Murtoluku
2  {
3  public:
4    Murtoluku(int kokonaisarvo); // Myös tyyppimuunnos
       ⋮
6  private:
7    int osoittaja_;
8    int nimittaja_;
9  };
       ⋮
10 Murtoluku::Murtoluku(int kokluku) : osoittaja_(kokluku), nimittaja_(1)
11 {
12 }
```

LISTAUS 7.11: Esimerkki rakentajasta tyyppimuunnoksena

luku. C++ osaa nyt automaattisesti käyttää tätä rakentajaa sekä eksplisiittisenä että implisiittisenä tyyppimuunnoksena. Kun tyyppimuunnos suoritetaan, ohjelma luo uuden nimettömän murtoluvun rakentajan avulla ja käyttää sitä tyyppimuunnoksen "lopputuloksena". Kääntäjä pitää myös huolen siitä, että tämä väliaikaisolio tuhotaan automaattisesti, kun sitä ei enää tarvita. Tässä suhteessa tyyppimuunnosten väliaikaisoliot käyttäytyvät täsmälleen samoin kuin paluuarvon välityksessä käytetyt väliaikaisoliot (aliluku 7.3).

Rakentajamuunnosta voi käyttää kaikkialla missä normaaliakin tyyppimuunnosta. Esimerkiksi muunnos `static_cast<Murtoluku>(3)` tuottaa lukua kolme vastaavan murtolukuolion luokan rakentajan avulla. Myös vanhat muunnossyntaksit `Murtoluku(3)` ja `(Murtoluku)3` toimivat. Näistä ensimmäinen muistuttaa jo syntaksinsakin puolesta rakentajan kutsumista. Myös implisiittiset tyyppimuunnokset toimivat normaalisti. Esimerkiksi koodissa

```
void kasittele(Murtoluku const& m);
     ⋮
kasittele(5);
```

kääntäjä huomaa, että funktion kutsumiseksi kokonaisluku 5 pitää muuttaa murtoluvuksi. Se tuottaa tällaisen murtoluvun rakentajan

avulla ja välittää funktiolle viitteen luotuun väliaikaisolioon. Funktiokutsun jälkeen väliaikaisolio tuhotaan.

Rakentajatyyppimuunnoksen estäminen

Rakentajan käyttö tyyppimuunnoksena saattaa aluksi tuntua kätevältä, mutta kun tämä ominaisuus lisättiin C++:aan, huomattiin pian, että ominaisuudella oli epätoivottuja sivuvaikutuksia. Ongelmana on, että kääntäjän kannalta *kaikki* yksiparametriset rakentajat ovat mahdollisia tyyppimuunnoksia. On kuitenkin paljon tilanteita, joissa rakentajan suorittamaa uuden olion luomista ei mitenkään järkevästi voi tulkita tyyppimuunnokseksi.

Jos ohjelmassa esimerkiksi määritellään oma taulukkotyyppi, sen rakentaja voi hyvinkin saada parametrinaan tiedon siitä, kuinka monen alkion taulukko halutaan luoda. Kuitenkaan ei ole mitenkään järkevää ajatella, että kyseessä olisi tyyppimuunnos kokonaisluvusta taulukoksi. Tässä tapauksessa rakentajan saama parametri on vain olion luomiseen tarvittava lisätieto eikä rakentaja määrää minkäänlaista tyyppimuunnosta. Kääntäjä ei kuitenkaan tätä tiedä ja saattaa käyttää taulukkoluokan rakentajaa myös *implisiittisenä* tyyppimuunnoksena tilanteissa, joissa (esimerkiksi kirjoitusvirheen vuoksi) kokonaisluvusta täytyisi tehdä taulukko. Tämä johtaa virheisiin, joiden löytäminen saattaa olla hyvin hankalaa.

Myöhemmin C++:ssa on yritetty paikata syntynyttä virhelähdettä lisäämällä kieleen avainsana **explicit**. Jos yksiparametrisen rakentajan edessä on luokan esittelyssä sana **explicit**, *kyseistä rakentajaa ei käytetä implisiittisenä tyyppimuunnoksena.* Käyttäjä voi kuitenkin halutessaan pyytää tyyppimuunnosta normaalilla tyyppimuunnossyntaksilla, mutta tämä tuskin tapahtuu vahingossa. Edellä mainitun taulukkotyypin rakentaja olisi esimerkiksi syytä varustaa **explicit**-sanalla seuraavaan tapaan:

```
class Taulukko
{
public:
  explicit Taulukko(int koko); // EI implisiittinen tyyppimuunnos
      ⋮
};
```

Vaikka **`explicit`**-sana ratkaiseekin syntyneen ongelman, se on edelleen varsin hankala käyttää, koska ohjelmoijan täytyy erikseen muistaa merkitä ne rakentajat, joita hän *ei* halua käyttää tyyppimuunnoksena. Kielen kannalta avainsana “implicit” olisi saattanut olla parempi, mutta se olisi aiheuttanut epäyhteensopivuutta kielen vanhojen ja uusien murteiden välillä.

Muunnosjäsenfunktiot

Rakentajien käyttö tyyppimuunnoksena tekee mahdolliseksi sen, että muita tyyppejä voi muuntaa ohjelmoijan kirjoittamaksi tyypiksi. Tyyppimuunnoksen tekeminen toiseen suuntaan ei onnistu samalla menetelmällä, koska ohjelmoija ei tietenkään voi kirjoittaa uusia rakentajia esimerkiksi **`int`**-tyypille tai kaupallisista kirjastoista löytyville tyypeille.

Tyyppimuunnos omasta tyypistä johonkin toiseen tyyppiin tapahtuu erityisten **muunnosjäsenfunktioiden** (*conversion member function*) avulla. Nämä ovat parametrittomia vakiojäsenfunktioita, joiden “nimi” muodostuu avainsanasta **`operator`** ja lisäksi muunnoksen kohdetyypistä. Esimerkiksi kokonaisluvuksi muuntavan muunnosjäsenfunktion nimi on **`operator int`**. Muunnosjäsenfunktio palauttaa paluuarvonaan muunnoksen lopputuloksen. Jäsenfunktion esittelyssä ei paluuarvoa merkitä näkyviin, koska se käy ilmi jo nimestä.

Listauksessa 7.12 seuraavalla sivulla on esimerkki murtolukuluokan muunnosjäsenfunktiosta, jonka avulla murtoluvun voi muuntaa liukuluvuksi. Kyseisen määrittelyn jälkeen kääntäjä osaa käyttää muutosta sekä implisiittisenä että eksplisiittisenä muunnoksena. Jos siis `m` on murtolukuolio, niin muunnokset `static_cast<double>(m)` ja `double d = m;` toimivat normaalisti.

Implisiittiset tyyppimuunnokset ovat tietysti mahdollinen virhelähde muunnosjäsenfunktioidenkin tapauksessa. Jostain syystä **`explicit`**-avainsanaa *ei* voi käyttää muunnosjäsenfunktioiden yhteydessä luomaan muunnoksia, joita voisi käyttää vain eksplisiittisesti. Jos tällaista halutaan, luokkaan täytyy kirjoittaa tavallinen jäsenfunktio, jota kutsutaan tyyppimuunnoksen sijaan.

```
1  class Murtoluku
2  {
3  public:
        ⋮
5    operator double() const; // Muunnosjäsenfunktio
        ⋮
9  };
        ⋮
13 Murtoluku::operator double() const
14 {
15   return static_cast<double>(osoittaja_)/static_cast<double>(nimittaja_);
16 }
```

LISTAUS 7.12: Esimerkki muunnosjäsenfunktiosta

7.5 Rakentajat ja struct

C++:ssa luokkia voi käyttää aivan samoissa paikoissa kuin kielen sisäänrakennettuja tyyppejäkin. Niinpä on täysin normaalia tehdä **struct**-tietorakenteita, joiden tietokenttinä on olioita, kuten päiväyksiä, `string`-merkkijonoja ja niin edelleen. Listauksessa 7.13 on esimerkkinä tällainen tietorakenne.

Kun **struct**-tietorakenne luodaan, sisältyy tähän kaikkien **struct**in kenttien luominen. Luokkatyyppiä olevilla kentillä tämä tarkoittaa kenttien rakentajien kutsumista. Jos tietorakenne luodaan “normaaliin tapaan” syntaksilla `Henkilotiedot henk;`, kutsutaan kentille automaattisesti oletusrakentajia, koska mitään alustamiseen tarvittavia parametreja ei ole saatavilla.

Jo C-kielessä oli myös mahdollista alustaa **struct** syntaksilla, jos-

```
1 struct Henkilotiedot
2 {
3   string nimi;
4   Paivays syntymapvm;
5   int ika;
6 };
```

LISTAUS 7.13: **struct**-tietorakenne, jossa on olioita

sa aaltosuluissa luetellaan luomisen yhteydessä kenttien alkuarvot. Tämä on mahdollista myös C++:ssa. Mikäli kenttinä on luokkia, näiden alustamisessa on kaksi vaihtoehtoa. Mikäli luokalla on yksiparametrinen rakentaja, joka kelpaa implisiittiseksi tyyppimuunnokseksi, voi aaltosulkulistaan laittaa tälle rakentajalle kelpaavan parametrin. Tällöin kentän alustus tapahtuu tyyppimuunnosrakentajaa käyttäen. Toinen tapa on kutsua aaltosulkulistassa suoraan jotain luokan rakentajaa ja antaa sille tarvittavat parametrit.

Alla on esimerkki molemmista tapauksista:

```
Henkilotiedot henk = {"K. Dari", Paivays(1, 1, 1970), 33};
```

Kenttä `nimi` alustetaan **`char`***-merkkijonoliteraalista `string`-luokan yksiparametrista rakentajaa käyttäen. Sen sijaan kenttä `syntymapvm` alustetaan alkuarvoonsa erikseen `Paivays`-luokan rakentajaa kutsumalla.

Olioiden tapauksessa aaltosulkualustus ei ikävä kyllä ole välttämättä kovin tehokas ratkaisu. C++-standardi nimittäin määrittelee alustuksen niin, että olioiden tapauksessa ensin luodaan aaltosulkulistassa olevien tietojen perusteella *irralliset* väliaikaisoliot ja nämä sitten *kopioidaan* **`struct`**-tietorakenteen sisään. Tämä tapahtuu luomalla kentät kopiorakentajilla, joille annetaan luodut väliaikaisoliot parametreina. Esimerkissä luodaan ensin väliaikaiset merkkijono `string("K. Dari")` ja päiväys, ja kentät `nimi` ja `syntymapvm` alustetaan sitten näistä kopiorakentajien avulla. Tehokas kääntäjä saa optimoida väliaikaisoliot pois, jos se siihen kykenee, mutta kopiorakentajan on silti oltava olemassa.

Aaltosulkualustuksen rajoituksena on myös, että sitä voi käyttää vain, jos ollaan luomassa nimettyä paikallista tai globaalia muuttujaa. Sen sijaan syntaksia ei voi käyttää väliaikaisolion tai **`new`**'llä luodun **`struct`**in luomiseen. Tällaisia tilanteita varten on joskus kätevää kirjoittaa **`struct`**-tietorakenteellekin *rakentaja,* joka alustaa tietorakenteen kentät haluttuihin alkuarvoihin.

Listaus 7.14 seuraavalla sivulla näyttää esimerkin tällaisesta rakentajasta. Rakentaja koostuu pelkästä alustuslistasta, joka alustaa tietorakenteen kentät annettujen parametrien avulla. Poikkeuksellisesti koko rakentaja on kirjoitettu **`struct`**-määrittelyn sisälle, koska rakentaja ei sisällä mitään toiminnalisuutta ja tällä tavoin sen koodia ei tarvitse kirjoittaa erilleen `.cc`-tiedostoon. Kyseessä on saman-

```
1 struct Henkilotiedot
2 {
3   string nimi;
4   Paivays syntymapvm;
5   int ika;
6   // Rakentaja, sisältää VAIN kenttien alustuksen
7   Henkilotiedot(string const& n, int p, int k, int v, int i)
8     : nimi(n), syntymapvm(p, k, v), ika(i) {}
9 };
```

LISTAUS 7.14: **struct**, jolla on rakentaja

lainen tilanne kuin rajapintaluokkien virtuaalipurkajien yhteydessä aliluvussa 6.9.2.

Rakentajan sisältävän tietorakenteen luominen tapahtuu nyt aivan kuten olionkin. Se onnistuu millä tahansa seuraavista syntakseista:

```
Henkilotiedot henk("K. Dari", 1, 1, 1970, 33);
Henkilotiedot* hp =
  new Henkilotiedot("Dari", 1, 1, 1970, 33);
Henkilotiedot("K. Dari", 1, 1, 1970, 33); // Väliaikaisolio
```

Vaikka periaatteessa tietorakenteen rakentajan runkoon tai alustuslistaan voisi C++:ssa kirjoittaa koodia, ei tätä tyylillisesti missään nimessä pitäisi tehdä. Ohjelmissa on totuttu pitämään **struct**-tietorakenteita "tyhminä" tietovarastoina, joihin ei sisälly ylimääräistä toiminnalisuutta. Niinpä **struct**in rakentaja onkin syytä pitää vain syntaktisena temppuna, jolla helpotetaan kenttien alustamista. Periaatteessa C++ sallisi myös purkajien ja jäsenfunktioiden kirjoittamisen **struct**-tietorakenteille, mutta näitäkään ei kannata käyttää, koska ne tekisivät ohjelmasta vain vaikealukuisemman tavalliselle ohjelmoijalle. Jos tietorakenteeseen on todella tarve upottaa toiminnallisuutta, kannattaa siitä yleensä kirjoittaa luokka **struct**-tietorakenteen sijaan.

Luku 8

Lisää rajapinnoista

"There are things known and things unknown and in between are The Doors."

– Jim Morrison [The Doors FAQ, 2002]

Rajapinnan tehtävä on kertoa siihen liittyvästä komponentista (moduuli tai olio) käyttäjälle vastaus kysymykseen "Miten tämän on tarkoitus toimia?". Kun käyttäjä tietää vastauksen tähän kysymykseen, hän pystyy hyödyntämään rajapinnan määrittelemää palvelua välittämättä sen taakse kapseloidusta toteutuksesta. Samalla komponentilla voi olla useita erilaisia rajapintoja. Oliolla voi olla rajapinta (tai sen osa) erikseen vakio-olioille, "tavallisille" instansseille ja aliluokille.

Rajapinnan toteuttajan pitäisi pitää mielessään lause "Tekeekö toteutus *täsmälleen* sen, mitä rajapinnan määrittely sanoo?". Taito ja kokemus ovat tässä työssä korvaamattomia. Ohjelmointikieli sekä suunnittelu- ja tyylisäännöt voivat ainakin ohjata kohti oikein toimivaa lopputulosta.

8.1 Sopimus rajapinnasta

Sopimussuunnittelu (*Design By Contract*, [Meyer, 1997]) on rajapintasuunnittelun ja -toteutuksen menetelmä, jossa palveluiden ominaisuuksia kuvataan matematiikan keinoin. Lakimiesten viilaaman kir-

jallisen kahden osapuolen sopimuksen tapaan sopimussuunnittelussa ajatellaan syntyvän rajapinnan käyttäjän ja sen toteutuksen välille sopimus, jossa kummallakin osapuolella on tarkkaan määritellyt vastuunsa:

- **Rajapinnan toteutus** lupaa jokaisen rajapinnan palvelun osalta toimia tietyllä tavalla, *kun rajapintaa käytetään sovitulla tavalla*. Määrittelyssä on siis mukana tieto siitä, mitkä ovat rajapinnan sallittuja käyttötapoja (funktioiden parametrit, kutsujärjestykset, tietotyyppien arvorajat yms.).

- **Rajapintaa käyttävä ohjelmoija** lupaa käyttää palveluita *ainoastaan* niiden määrittelyn mukaisesti.

Sopimussuunnittelu on nimensä mukaisesti ennen kaikkea rajapintojen suunnittelua auttava ajattelutapa, joka pakottaa ja ohjaa sekä miettimään että dokumentoimaan rajapinnan huolellisesti. Toissijaisena hyötynä menetelmä yksinkertaistaa komponenttien toteuttamista. Tämä saavutetaan siten, että rajapinnan toteutuksen ulkopuolelle on sopimuksessa rajattu hankalat ja ei-toivotut käyttötavat, jolloin niihin ei tarvitse toteutuksessa varautua. Voidaan ajatella että “villin lännen” rajapintaa saa käyttää miten huvittaa ja toteutuksen pitäisi osata toimia kaikissa tilanteissa jotenkin “oikein” (minimissään laadukas koodi ei ainakaan kaadu kummallisissa tilanteissa). Sopimussuunnittelussa jaetaan vastuuta toiminnasta kutsujan ja toteutuksen kesken, jolloin kokonaisohjelmakoodimäärä on yleensä pienempi kuin kaikkeen varautuneissa toteutuksissa.

8.1.1 Palveluiden esi- ja jälkiehdot

Rajapinnan yksittäisen palvelun sopimus tehdään määrittelemällä jokaiselle palvelulle **esiehto** (*precondition*, P) ja **jälkiehto** (*postcondition*, Q). Molemmat ovat (predikaattilogiikan) loogisia lausekkeita, joista esiehdon tulee olla totta ennen palvelun käynnistämistä ja jälkiehdon palvelun suorittamisen jälkeen:

$$\{P\}\ \texttt{palvelu()}\ \{Q\}$$

Tämän “oikeellisuuskaavan” (*correctness formula*) mukaisesti palvelu lupaa, että kaikki funktion `palvelu` suoritukset, joiden alkaessa ehto P on totta, päättyvät tilaan, jossa ehto Q on totta. Jos ehto P ei

toteudu, niin palvelu toimii määrittelemättömästi ja jälkiehdon ei tarvitse toteutua (haluttua palvelua ei saada).

Yksinkertaisimmillaan ehdoilla voidaan rajata parametrien arvoalueita:

{ p >= kirja.lainauspäivä }
`kirja.palauta(palautuspäivä p)`
{ kirja.tila = PALAUTETTU }

Käytettäessä predikaattilogiikan ominaisuuksia voidaan kertoa esimerkiksi rutiinista, joka järjestää taulukon, joka ei sisällä yhtäkään tyhjää alkiota:

$$\{\forall i \mid 1 \leq i \leq n : ARRAY[i] \neq \text{TYHJÄ}\}$$
`qsort(ARRAY)`
$$\{\forall i \mid 1 \leq i < n : ARRAY[i] \leq ARRAY[i+1]\}$$

Sopimussuunnittelun tärkein ominaisuus on palvelun kutsujan ja toteuttajan vastuiden määrittely — samalla suunnittelun suurin vaikeus on päättää, mitkä ominaisuudet kuuluvat vastuurajan millekin puolelle.

- **Kutsuja.** Palvelun kutsujan vastuulla on pitää huoli siitä, että esiehto toteutuu. Jos kirja yritetään palauttaa päivämäärällä joka on ennen kirjattua lainauspäivää, niin palvelun mukainen sopimus ei ole voimassa, ja suoritus voi tehdä mitä tahansa. Ohjelman *testausvaiheessa* voidaan esiehtoja tarkastaa, mutta päämääränä on, että näitä tarkastuksia ei ole enää lopullisessa ohjelmistossa mukana (huolellinen testaus on löytänyt esiehdon rikkovat palvelun kutsut).

- **Toteuttaja.** Palvelun toteuttajan vastuulla on pitää huoli siitä, että palvelun suorituksen jälkeen jälkiehto on aina voimassa (kunhan esiehto on ollut voimassa). Jos sopimuksessa on määritelty rutiinin parametripäiväyksen olevan jotain päivämäärää myöhemmän, niin palvelun toteutuksessa ***ei enää tehdä tarkastusta***, joka on määritelty kutsujan vastuulle. Jos palvelu ei pysty toteuttamaan sopimuksen mukaista palvelua (jälkiehto ei täyty) kyseessä on virhetilanne, joka on jollain tavalla ilmaistava. Tämä tehdään yleensä poikkeusten (luku 11) avulla.

Yleiskäyttöisten komponenttien rajapinnoissa on usein vaikea päättää, minkälainen sopimus halutaan muodostaa kutsujan ja toteutuksen välille. Joskus voi olla tarkoituksenmukaista tarjota samasta perustoiminnallisuudesta sopimukseltaan erilaisia versioita.

Yksi esimerkki tällaisesta toiminnasta on `vector`-taulukon indeksointi, josta löytyy kaksi versiota. Operaatio `at()` hyväksyy parametrinaan minkä tahansa kokonaisluvun ja tarkistaa toteutuksessaan osuuko tämä luku indeksinä tulkittuna taulukon sisälle. Jos indeksi on kelvollinen, niin kyseessä oleva alkio palautetaan, muutoin kutsujalle kerrotaan virheestä (poikkeusmekanismilla). Toinen versio indeksoinnista on hakasulkuoperaattori (`v[i]`), joka ei suorita indeksin laillisuustarkistusta. Sopimussuunnittelun kannalta tämä versio on määritellyt kutsujan vastuulle (esiehdoksi), että operaatiota kutsutaan ainoastaan laillisilla taulukon indekseillä. Jos kutsuja ei täytä esiehtoa, niin toteutus tekee määrittelemättömän operaation (joka ohjelmassa näkyy esimerkiksi sen kaatumisena tai muistin roskaantumisena).

8.1.2 Luokkainvariantti

Luokkien tapauksessa voidaan palveluiden esi- ja jälkiehtojen lisäksi määritellä pysyväisväittämä eli **invariantti** (*invariant*), joka kertoo luokan olion ylläpitämän ehdon palvelukutsujen välillä. Esimerkiksi:

```
class KirjastonKirja {
  // Invariantti: sijaintitieto on aina
  // LAINASSA, PAIKALLA, HUOLTO tai POISTETTU
...
```

Esimerkin `KirjastonKirja`-luokan olioiden tilan tiedetään olevan aina invariantin mukainen silloin, kun ohjelman suoritus ei ole keskellä jotain luokan tarjoamaa palvelua.

Luokkainvariantin on oltava voimassa heti olion synnyttyä. Jos oliota luotaessa kutsutaan alustusrutiinia (kuten C++:n rakentajajäsenfunktio), invariantin on oltava voimassa, kun tämä rutiini on lopettanut suorituksensa. Tämän jälkeen invariantin on oltava voimassa aina ennen ja jälkeen jokaista julkisen rajapinnan rajapintafunktion kutsua:

$$\{\text{LUOKKAINVARIANTTI} \wedge P\} \quad \texttt{olio.palvelu()} \quad \{\text{LUOKKAINVARIANTTI} \wedge Q\}$$

Kun suoritus on jäsenfunktion koodissa olion "sisällä", puhutaan epästabiilista tilasta, jossa luokkainvariantti saa väliaikisesti olla epätosi. Tämän säännön noudattaminen ja tarkastaminen tulee hankalaksi silloin kun jäsenfunktio kutsuu toisia jäsenfunktioita osana omaa toiminnallisuuttaan.

8.1.3 Sopimussuunnittelun käyttö

Yksinkertaisissa esimerkeissä kauniilta näyttävä sopimussuunnittelu ei valitettavasti skaalaudu varsinkaan ohjelmiston ylimmän tason moduulien rajapintoihin. Esi- ja jälkiehtojen tarkkaan määrittelyyn kuuluu usein niin paljon informaatiota, että sen kuvaaminen matemaattisen tarkasti on ainakin tällä hetkellä liian työlästä. Formaaleihin määrittely- ja suunnittelumenetelmiin liittyvää tutkimusta tehdään paljon, ja sitä sovelletaan useissa erityisen suurta varmuutta vaativissa ohjelmistoprojekteissa. Tavallisimmissa ohjelmistoprojekteissa matemaattista määrittelyä ei kuitenkaan yleensä juuri käytetä. Sopimussuunnittelun periaatteiden tunteminen ja jopa osittainenkin noudattaminen on tietysti pelkkää sanallista rajapintakuvausta eksaktimpi menetelmä.

Olio-ohjelmointi aiheuttaa myös omat hankaluutensa sopimussuunnitteluun. Periytymisen yhteydessä rajapinnan sopimuksen pitäisi usein osata sanoa jotain myös aliluokan rajapinnasta, joka usein on hyvin hankalaa. Joskus kantaluokka ei tiedä edes karkeasti minkälaisia versioita siitä on tarkoitus periyttää, jolloin liian tiukasti määritellyt rajapintasopimukset voivat pahimmillaan tehdä periytettyjen versioiden tekemisen mahdottomiksi siten, että kantaluokan rajapintasopimus pidetään voimassa.

8.1.4 C++: luokan sopimusten tarkastus

Ohjelmiston kehitysvaiheessa on hyödyllistä tarkastaa, että rajapinnoille määriteltyjä sopimuksia noudatetaan. Väärinymmärryksistä tai huonosta dokumentoinnista johtuvat sopimuksen vastaiset kutsut saadaan heti näkyville ohjelmiston testausvaiheessa. Ylimääräiset testit hidastavat ohjelman toimintaa, mutta testausvaiheessa sillä ei useinkaan ole merkitystä. Poikkeuksena ovat reaaliaikavaatimuksia omaavat ohjelmistot, joissa ei voida käyttää testeissäkään ylimää-

räisiä tarkastuksia, jos niiden aiheuttama ylimääräinen prosessointi vaikuttaa ohjelmiston ajoituksiin.

assert-makro

Ohjelmissa olevilla sopimustarkastuksilla on pitkät perinteet jo ajalta ennen olio-ohjelmoinnin yleistymistä. C-kielessä on määriteltynä **varmistusrutiini** `assert` [Kerninghan ja Ritchie, 1988], joka ilmoittaa virheestä ja pysäyttää ohjelman suorituksen, jos sen parametrina oleva lauseke on arvoltaan epätosi. Kun ohjelmiston julkaisuversio käännetään siten, että esikääntäjäsymboli `NDEBUG` on määriteltynä, `assert`-makron arvo asettuu tyhjäksi, jolloin nämä ***kehitysvaiheen tarkastukset eivät ole mukana lopullisessa ohjelmistossa***. Tämä käytäntö (assert-testit ovat mukana vain ohjelman testausvaiheessa) noudattaa sopimussuunnittelun periaatetta, jonka mukaan oikein toimivassa ohjelmassa sekä rajapinnan kutsu että toteutus noudattavat *aina* sopimusta.

Koska `assert` on määritelty makroksi, se *ei* ole C++:n `std`-nimiavaruuden sisällä. Rajapintafunktion alussa voidaan testauksessa varmistaa kutsujan noudattavan sopimusta funktion parametreista:

```
#include <cassert>
     ⋮
int palvelu(int a, int b)
{
  assert( a < 2 && b > 40 );
     ⋮
```

Funktion väärä käyttö voi näkyä testaajalle esimerkiksi seuraavassa muodossa:

```
assertkoe.cc:4: failed assertion 'a < 2 && b > 40'
(program aborted)
```

Tässä `assertkoe.cc` on lähdekooditiedoston nimi. C++-standardi jättää tarkoituksella määrittelemättä tulostettavan virheen muodon. Jossain ympäristössä tällainen virheilmoitus voidaan ilmaista esimerkiksi käyttöliittymän virheikkunalla.

Koska `assert`-tarkastukset eivät ole mukana lopullisessa ohjelmakoodissa, on oltava tarkkana, että käytetty ***varmistusehto ei sisällä***

ohjelman toiminnallisuutta:

```
1   Paivays* pNyt = 0;
2   assert(pNyt = Paivays::NytOsoitin()); // Sijoitus assertissa!
3   pNyt->Tulosta();
```

Vaikka assert-makron parametrina oleva sijoitus (rivi 2) saakin C++:ssa arvon, joka tulkitaan myös totuusarvoksi, kyseessä on silti väärä tapa käyttää varmistusrutiinia.[ϒ] Edellinen koodinpätkä voi toimia täysin oikein ohjelmiston testausvaiheessa, mutta kun lopullisessa versiossa assert-makro määritellään tyhjäksi, rivin 2 sijoitus jää kokonaan suorittamatta ja koodin toiminta muuttuu.

C-kielen assert on määritelty siten, että ohjelman suoritus keskeytetään kutsumalla kielen funktiota abort. Tätä funktiota ei pidetä C++:ssä suositeltavana, koska sitä kutsuttaessa ei suoriteta mitään lopetustoimenpiteitä. Erityisesti ohjelmassa kutsuhetkellä olevien olioiden purkajat jäävät suorittamatta (niissä voi olla resurssien vapautukseen liittyviä toimintoja). C++:ssä suositeltavampi tapa on käyttää poikkeusmekanismia (luku 11) myös “assertiovirheen” ilmaisemiseen. Listauksessa 8.1 seuraavalla sivulla on esimerkki poikkeuksilla toteutetusta Assert-makrosta C++:lla.

Luokkainvariantti

Koska luokkainvariantin tarkastuksessa on tarkoituksena tarkastaa luokan vastuualueen sopimus jokaisen rajapintafunktion suorituksen jälkeen, kannattaa tämä tarkastus kirjoittaa omaksi jäsenfunktiokseen, jota kutsutaan aina muiden jäsenfunktioiden alussa ja lopussa. (Myös alussa, jotta voidaan varmistua siitä, että invariantin määräämä sopimus on voimassa myös rutiinin suorituksen alkaessa, katso ohjelmalistaus 8.2 sivulla 248).

Tässä esitetty suoraviivainen toteutus ei huomioi mitenkään sitä tilannetta, että invariantti saa olla epätosi jos tarkistuksen kohteena olevaa jäsenfunktiota on kutsuttu toisesta (saman olion) jäsenfunktiosta. Tällainen kutsuketjun seuraaminen mutkistaisi ohjelmakoodia jonkin verran, ja sen toteuttamiseen ei C++:ssä valitettavasti ole valmiita apukeinoja.

[ϒ] Hyvin tyypillinen virhe C++:ssa on käyttää yhtäsuuruutta tarkoitettaessa kielen (huonosti valittua) sijoitusmuotoa [assert(x = 7) vs. assert(x == 7)].

```
 1 #ifndef ASSERT_HH
 2 #define ASSERT_HH
 3
 4 #include "varmistuspieleen.hh"          /* poikkeusluokan esittely */
 5 #include <iostream>
 6
 7 #ifdef NDEBUG
 8 #define Assert(x) /* tyhja */
 9 #else
10 #define Assert(x) Assert_toteutus( x, #x, __FILE__, __LINE__ )
11 #endif
12
13 inline void Assert_toteutus( bool varmistus, char const* lauseke,
14                              char const* tiedosto,
15                              unsigned int rivinumero )
16 {
17   if( varmistus == false ) {
18     std::cerr << tiedosto << ":" << rivinumero << ":";
19     std::cerr << "Varmistus epäonnistui: " << lauseke << std::endl;
20     throw VarmistusPieleen(lauseke, tiedosto, rivinumero);
21   }
22 }
23
24 #endif /* ASSERT_HH */
```

LISTAUS 8.1: Varmistusrutiinin toteutus

Koska C++ ei automaattisesti tue invarianttien tarkastusta, niiden käyttö on täysin ohjelmoijan vastuulla. Tämä tulee ottaa huomioon myös periytymishierarkioissa, joissa on pidettävä huoli siitä, että aliluokan jäsenfunktiot tarkastavat myös kantaluokan invariantin säilymisen:

```
// Kun järjestetty taulukko on periytetty luokasta Taulukko
inline void JärjestettyTaulukko::Invariantti()
{
#ifndef NDEBUG
  Taulukko::Invariantti();
     ⋮
#endif
}
```

```
1  inline void JarjestettyTaulukko::Invariantti()
2  {
3  #ifndef NDEBUG
4    // Invariantti: alkiot ovat aina suuruusjärjestyksessä, siten että
5    // indeksissä 1 on pienin alkio ja indeksissä KOKO suurin.
6    for( int i = 1; i < KOKO; i++ )
7    {
8      if( alkio[ i ] > alkio[ i+1 ] )
9        throw JarjestettyTaulukko::InvarianttiRikottu();
10   }
11 #endif
12 }
13
14 void JarjestettyTaulukko::EtsiJaMuutaAlkio(
15   Alkio const& etsittava, Alkio const& korvaava )
16 {
17   Invariantti();
18   // Jäsenfunktion toteutus
19   Invariantti();
20 }
```

LISTAUS 8.2: Esimerkki luokkainvariantin toteutuksesta jäsenfunktiona

8.2 Luokan rajapinta ja olion rajapinta

Luokkasuunnittelussa tulee silloin tällöin vastaan tilanne, jossa jokin luokan vastuualueeseen kuuluva asia ei oikeastaan kuulu minkään luokan olion tehtäviin vaan ikään kuin "koko luokalle", toisaalta kaikille, toisaalta ei millekään oliolle. Esimerkkejä tällaisista luokan "yhteisistä" asioista ovat esim. päiväysluokalla taulukko, jossa pidetään muistissa kuukausien pituudet, tai merkkijonoluokalla tieto siitä, mikä on merkkijonojen maksimipituus. Samoin ohjelman testausvaiheessa saattaisi olla mukavaa pitää kirjaa siitä, montako jonkin luokan oliota ohjelman ajon aikana on luotu, sekä tarjota funktio, jolla tuon tiedon saa tulostettua.

Tällaiset luokalle yhteiset asiat pitäisi tietysti jotenkin sitoa itse luokkaan ja kapseloida niin, että vain käyttäjälle tarkoitettu "luokan rajapinta" näkyy ulospäin. Tämän saavuttamiseen eri oliokielissä on käytetty erilaisia mekanismeja. Tässä teoksessa tutustutaan lyhyesti kahteen eri mekanismiin: Smalltalkin metaluokkiin ja luokkaolioihin sekä C++:n luokkafunktioihin ja -muuttujiin. Jotkin ohjelmointi-

kielet sisältävät myös kompromisseja näiden kahden välillä, esimerkiksi Javan luokkaolioiden käsite on yhdistelmä Smalltalkin ja C++:n mekanismeja.

8.2.1 Metaluokat ja luokkaoliot

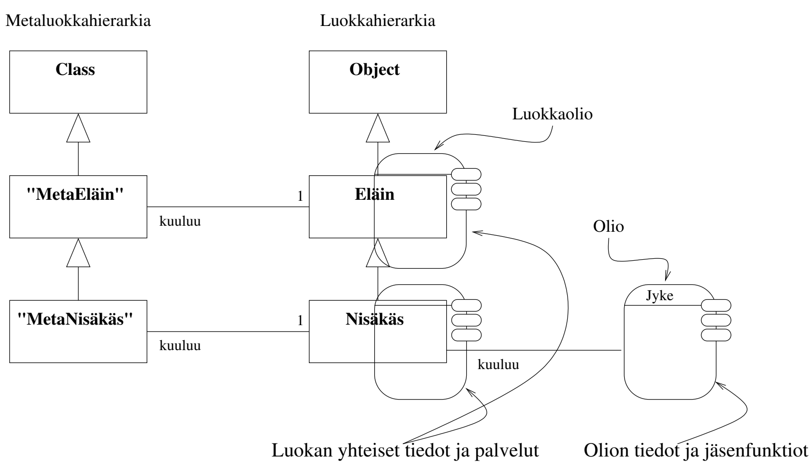
Puhtaasti oliopohjainen lähestymistapa luokan yhteisiin asioihin on, että luokan yhteisen datan ja operaatioiden täytyy olla jonkin olion attribuutteja ja operaatioita. Tämä johtaa luontevasti ajatukseen, että jokaista luokkaa vastaa yksikäsitteinen olio, jonka vastuulla luokan yhteiset asiat ovat. Vielä vähän pidemmälle vietynä saadaan aikaan oliomalli, jossa *jokainen luokka on olio*, **luokkaolio** (*class object*), joka hoitaa kaikki koko luokan vastuulla olevat asiat.

Smalltalkissa on otettu juuri tämä lähestymistapa luokkiin, ja se on itse asiassa oleellinen osa kielen toteutusta. Uusien olioiden luominen on tietysti yksi asia, joka kuuluu luokan vastuualueeseen. Niinpä Smalltalkin operaatio `new`, jolla luodaan uusi olio, on itse asiassa luokkaa vastaavan luokkaolion jäsenfunktio, joka luo uuden olion ja palauttaa sen paluuarvonaan.

Smalltalkissa luokkaolion nimi on sama kuin itse luokan nimi ja luokkaoliota voi käyttää kaikkialla missä normaaleitakin olioita. Luokkaolion voi esimerkiksi antaa parametrina toiselle oliolle, joka käyttää sitä luomaan luokasta uusia olioita.

Ongelmaksi jää, että jokaisen olion pitää kuulua johonkin luokkaan. Jos kerran luokkakin on vain olio, mihin luokkaan se sitten kuuluu? Smalltalkin ratkaisu on sanoa, että jokainen luokkaolio kuuluu omaan **metaluokkaansa** (*metaclass*), jonka ainoa olio se on. Kaikki metaluokat puolestaan on periytetty luokasta `Class`, joka sisältää kaikki kaikille luokille yhteiset ominaisuudet. Näin metaluokat muodostavat keskenään luokkahierarkian, joka on rakenteeltaan täsmälleen samanlainen kuin tavallisten luokkien muodostama hierarkia. Kuvassa 8.1 seuraavalla sivulla on esimerkkinä osa metaluokkien ja tavallisten luokkien muodostamaa hierarkiaa.

Smalltalkin metaluokkahierarkia antaa mahdollisuuden varsin mielenkiintoisiin olioratkaisuihin, ja sen avulla voidaan saada aikaan mm. samanlaisia rakenteita kuin C++:n mallien (aliluku 9.5) avulla. Tässä teoksessa ei aiheeseen kuitenkaan keskitytä enempää, mutta asiasta kiinnostuneelle sopivaa kirjallisuutta on mm. “An Introduc-



KUVA 8.1: Luokka- ja metaluokkahierarkiaa Smalltalkissa

tion to Object-oriented Programming, 3rd edition" [Budd, 2002, luku 25].

8.2.2 C++: Luokkamuuttujat

C++:n kehittäjät päättivät, että metaluokista ja luokkaolioista saatava hyöty oli turhan pieni niiden toteutusvaivaan nähden. Tämän vuoksi C++:ssa on käytössä erilainen mekanismi luokan yhteisiä asioita varten.

Jos jokin muuttuja on luonteeltaan sellainen, että se kuuluu koko luokalle eikä vain yhdelle oliolle (siis muuttujan halutaan olevan luokan kaikille olioille yhteinen), muuttuja esitellään luokan esittelyssä **luokkamuuttujana** (*static data member*). Luokkamuuttujan esittely on muuten samanlainen kuin jäsenmuuttujankin, mutta esittely al-

kaa avainsanalla **static**:

```
class X
{
   ⋮
  int jasenmuuttuja_;
  static int luokkamuuttuja_;
};
```

Luokkamuuttuja on luonteeltaan hyvin lähellä normaalia globaalia muuttujaa: sen elinkaari on ohjelman alusta ohjelman loppuun saakka. Luokkamuuttuja on siis olemassa, vaikka luokasta ei olisi vielä luotu ainuttakaan oliota. Koska luokkamuuttuja ei ole minkään olion oma, sen alustamista ei voi tehdä rakentajan alustuslistassa jäsenmuuttujien tapaan, vaan jossain kooditiedostossa täytyy olla erikseen luokkamuuttujan määrittely, jonka yhteydessä muuttuja alustetaan:

```
// Yleensä samassa kooditiedostossa kuin jäsenfunktioiden koodi
int X::luokkamuuttuja_ = 2;
```

Luokan omassa koodissa luokkamuuttujaan voi viitata aivan kuten jäsenmuuttujaankin, suoraan sen nimellä. Luokan ulkopuolelta niihin voi viitata syntaksilla `Luokka::lmuuttuja` (esimerkissä `X::luokkamuuttuja_`). Toinen (harvemmin käytetty) tapa on viitata luokkamuuttujiin luokan olion kautta ikään kuin luokkamuuttuja olisi jäsenmuuttuja:

```
X xolio;
int arvo = xolio.luokkamuuttuja_;
```

Luokkamuuttujille pätevät hyvin pitkälle samat tyylisäännöt kuin jäsenmuuttujillekin, mm. luokkamuuttujat olisi hyvä pitää luokan private-puolella. Tähän on kuitenkin yksi yleinen poikkeus — luokkavakiot. Varsin usein luokkaan liittyy epälukuinen määrä rajoituksia, maksimiarvoja yms. lukuja, joihin perinteisesti olisi käytetty **#define**-vakioita tai globaaleja vakioita. Huomattavasti parempi käytäntö on kuitenkin upottaa tällaiset vakiot luokan sisään, jolloin on selvää, mitä osaa ohjelmasta ne koskevat. Samalla myös nimikonfliktien vaara pienenee.

Luokkavakioita saa tehtyä yksinkertaisesti luomalla luokkamuuttuja, joka on määritelty vakioksi **const**-määreellä. Ongelmaksi jäävät vain tilanteet, joissa tällaisen vakion tulee olla käännösaikainen vakio. Mikäli vakion alustaminen jätetään mielivaltaiseen kooditiedostoon, kääntäjä ei löydä sitä eikä pysty käyttämään vakion arvoa käännösaikana. Tämä ratkaisemiseksi C++-standardiin otettiin mukaan mahdollisuus alustaa *kokonaislukutyyppiset* luokkavakiot suoraan luokan esittelyssä. Tyypillisesti tällaiset vakiot sijoitetaan luokan public-osaan, koska niiden käyttötarkoituksena on juuri näkyä luokasta ulospäin:

```
class Y
{
public:
  static int const MAX_PITUUS = 80;
    ⋮
};
```

Vaikka tällaiset luokkavakiot alustetaankin luokan esittelyssä, ne täytyy silti edelleen määritellä jossain kooditiedostossa (mutta alustusta ei enää toisteta):

```
// Yleensä samassa kooditiedostossa kuin jäsenfunktioiden koodi
int const Y::MAX_PITUUS;
```

Luokkavakioita voi käyttää luokan omassa koodissa tavallisten luokkamuuttujien tapaan.

8.2.3 C++: Luokkafunktiot

Luokkafunktiot (*static member function*) muistuttavat jäsenfunktioita samalla tavalla kuin luokkamuuttujat jäsenmuuttujia. Luokkafunktiot edustavat sellaisia luokan palveluja ja operaatioita, jotka eivät kohdistu mihinkään yksittäiseen olioon, vaan “koko luokkaan”. Tällaisia operaatioita ovat mm. luokkamuuttujien käsittely, mutta on tietysti mahdollista että luokan vastuualueeseen kuuluu muitakin “luokanlaajuisia” operaatioita. Luokkafunktiot esitellään luokan esittelyssä

lisäämällä niiden eteen sana **static**:

```
class X
{
public:
  int jasenfunktio();
  static int luokkafunktio();
    ⋮
};
```

Luokkafunktioiden määrittely kooditiedostossa puolestaan ei eroa jäsenfunktion määrittelystä. Ainoat rajoitukset ovat, että koska luokkafunktio ei kohdistu mihinkään olioon, sen koodissa ei voi viitata jäsenmuuttujiin, jäsenfunktioihin eikä **this**-osoittimeen. Sen sijaan luokkafunktion koodissa voi vapaasti käyttää luokkamuuttujia ja toisia luokkafunktioita.

Listaus 8.3 seuraavalla sivulla sisältää esimerkkinä luokan, joka pitää kirjaa siitä, montako luokan oliota ohjelman aikana on luotu ja tuhottu. Laskurit on talletettu luokkamuuttujina `luotu_` ja `tuhottu_` ja ne pystyy tulostamaan luokkafunktion `tulosta_tilasto` avulla.

8.3 Tyypit osana rajapintaa

Listauksessa 8.4 sivulla 255 on osa tyypillistä päiväysluokan rajapintaa. Aiemmin tällaista rajapintaa mainostettiin erityisesti sen vuoksi, että sen sanottiin piilottavan luokan toteutuksen käyttäjältä, jolloin luokan toteutusperiaatteen muuttaminen on helppoa.

Kieltämättä listauksen päiväysluokassa on nyt mahdollista esittää päiväys sisäisesti mitä erilaisimmilla tavoilla, mutta luokan rajapinta on myös lupaus. Rajapinnalla luokka lupaa, että sitä voi käyttää rajapinnan mukaan, ja tästä lupauksesta on pidettävä kiinni. Toisaalta tämä lupaus sitoo myös luokan mahdollisia toteutuksia, toisin sanoen muutettiinpa luokan toteutusta millä tavoin hyvänsä, niin rajapinnan on pysyttävä *täsmälleen* samana.

Kuvitellaanpa esimerkiksi, että päiväysluokkaa on onnistuneesti käytetty jo pitkään sukupuuohjelmassa ja kaikki on toiminut mainiosti. Myöhemmin huomataan, että periaatteessa mikään ei estä päiväyksen vuosiluvun menemistä negatiiviselle puolelle, jos suvun jo-

oliolaskuri.hh

```
1  #ifndef LASKIJA_HH
2  #define LASKIJA_HH
3
4  class Laskija
5  {
6  public:
7    Laskija(int luku);
8    ~Laskija();
       ⋮
9    static void tulosta_tilasto();
10 private:
11   int luku_;
       ⋮
12   static unsigned int luotu_;
13   static unsigned int tuhottu_;
14 };
15
16 #endif
```

oliolaskuri.cc

```
1  #include "oliolaskuri.hh"
2
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  Laskija::Laskija(int luku) : luku_(luku)
8  {
9    ++luotu_;
10 }
11
12 Laskija::~Laskija()
13 {
14   ++tuhottu_;
15 }
16
17 void Laskija::tulosta_tilasto()
18 {
19   cout << "Olioita luotu: " << luotu_ << endl;
20   cout << "Olioita tuhottu: " << tuhottu_ << endl;
21 }
22
23 unsigned int Laskija::luotu_ = 0;
24 unsigned int Laskija::tuhottu_ = 0;
```

LISTAUS 8.3: Esimerkki luokkamuuttujien ja -funktioiden käytöstä

```
1  class Paivays
2  {
3  public:
4    Paivays(unsigned int p, unsigned int k, unsigned int v);
5    ~Paivays();
6
7    void asetaPaiva(unsigned int paiva);
8    void asetaKk(unsigned int kuukausi);
9    void asetaVuosi(unsigned int vuosi);
10
11   unsigned int annaPaiva() const;
12   unsigned int annaKk() const;
13   unsigned int annaVuosi() const;
14
15   void etene(int n);
16   int paljonkoEdella(Paivays const& p) const;
17
18 private:
19   unsigned int paiva_;
20   unsigned int kuukausi_;
21   unsigned int vuosi_;
22 };
```

LISTAUS 8.4: Tyypillinen päiväysluokan esittely

takin haaraa voidaan seurata todella pitkään (todellisuudessa tällainen mahdollisuus lienee korkeintaan joillain kuningassuvuilla). Eipä hätää, ohjelma on tehty olio-ohjelmointia käyttäen, joten päiväysluokan sisäisen toteutuksen muuttaminen on helppoa. Ohjelman tekijä lupaa puhelimessa korjata ohjelman tunnissa ja alkaa tutkia koodia.

On kyllä totta, että luokan sisäisen toteutuksen muuttaminen on todella helppoa — ainoa muutos on muuttaa rivi 21 muotoon "`int vuosi_;`". Tämä ei kuitenkaan ratkaise ongelmaa, koska luokan rajapinnassa käytetään vuosilukujen käsittelyyn tyyppiä `unsigned int`.

Seuraava ratkaisuyritys on luonnollisesti muuttaa uusi tyyppi myös rajapintaan. Tämä vaatii jonkin verran tarkkuutta (jotta kaikista rajapinnan `unsigned int` -tyypeistä muutetaan vain ja ainostaan tarpeelliset), mutta on suhteellisen helppo huomata, että rakentajan ja `asetaVuosi`-jäsenfunktion `vuosi`-parametrin tyyppi pitää korjata, samoin `annaVuosi`-jäsenfunktion paluutyyppi. Helpotuksesta huokaisten ohjelmoija kääntää ohjelman uudelleen ja aloittaa testauksen.

Testaus osoittaa, että ohjelma toimii väärin!

Vikana on, että alkuperäinen päiväysluokan rajapinta vihjaa käyttäjälle varsin voimakkaasti, että päiväyksistä saatavia vuosilukuja saa ja tuleekin tallettaa **unsigned int** -tyyppisiin muuttujiin. Niinpä ohjelmasta saattaa löytyä paljonkin koodia, joka näyttää vaikkapa seuraavalta:

```
unsigned int syntymaVuosi = henkilo.syntymapvm.annaVuosi();
if (syntymaVuosi % 10 == 0)
{
   cout << "Syntynyt tasakymmenluvulla" << endl;
}
```

Kun nyt vuodet muutetussa luokassa palautetaankin **int**-tyyppisinä, tehdään ensimmäisellä rivillä tyyppimuunnos **int** ⇒ **unsigned int**. Mikäli syntymävuosi sattuu olemaan negatiivinen, sitä ei voi esittää etumerkittömällä luvulla, joten syntymaVuosi-muuttujaan tallettuu väärä arvo![♉]

Esimerkin kaltaisten tapausten vuoksi on tärkeä ymmärtää, että myös rajapinnan *tyypit* ovat osa rajapintaa. Tämä koskee niin parametrien tyyppejä kuin paluutyyppejäkin. Mikäli rajapinta määrää tyypin tietyksi, alkaa rajapintaa käyttävä koodi helposti luottaa tyypin säilymiseen samana. Mikäli alkuperäinen päiväysluokka olisi "tilaa säästääkseen" käyttänyt **unsigned char**-tyyppiä vuosiluvun välittämiseen, olisi vuosi 2000 -ongelma päässyt syntymään olio-ohjelmoinnista huolimatta.

Ratkaisu ongelmaan on toistaa tuttua kapselointiperiaatetta uudelleen: kapseloidaan rajapinnan tyypit omien tyyppinimien taakse, ja vaaditaan käyttäjiä käyttämään tarjottuja tyyppejä. Listaus 8.5 seuraavalla sivulla sisältää parannellun päiväysluokan esittelyn. Riveillä 5–8 esitellään kaikki luokan rajapinnassa käytetyt tyypit ja annetaan niille nimi. Tämän jälkeen kaikkialla luokassa käytetään päivänumeron yhteydessä tyyppiä PaivaNro jne.

[♉] C++-standardissa sanotaan, että negatiivinen luku muuntuu etumerkittömäksi luvuksi niin, että $\mathit{uusiluku} \equiv \mathit{vanhaluku} \pmod{2^n}$, missä n on etumerkittömän luvun bittien lukumäärä.

```
1  class Paivays
2  {
3  public:
4    // Luokan käyttämät tyypit
5    typedef unsigned char PaivaNro;
6    typedef unsigned char KkNro;
7    typedef unsigned long int VuosiNro;
8    typedef long int Erotus;
9
10   Paivays(PaivaNro p, KkNro k, VuosiNro v);
11   ~Paivays();
12
13   void asetaPaiva(PaivaNro paiva);
14   void asetaKk(KkNro kuukausi);
15   void asetaVuosi(VuosiNro vuosi);
16
17   PaivaNro annaPaiva() const;
18   KkNro annaKk() const;
19   VuosiNro annaVuosi() const;
20
21   void etene(Erotus n);
22   Erotus paljonkoEdella(Paivays p) const;
23
24 private:
25   PaivaNro paiva_;
26   KkNro kuukausi_;
27   VuosiNro vuosi_;
28 };
```

LISTAUS 8.5: Parannettu päiväysluokan esittely

Luokan käyttäjän tulee nyt käyttää nimettyjä tyyppejä:

```
Paivays::VuosiNro syntymaVuosi =
  henkilo.syntymapvm.annaVuosi();
if (syntymaVuosi % 10 == 0)
{
   cout << "Syntynyt tasakymmenluvulla" << endl;
}
```

Jos nyt tämän päiväysluokan kanssa tulee tarve sallia negatiiviset vuodet, on muutoksen tekeminen helppoa: muutetaan rivi 7 muotoon

```
typedef long int VuosiNro;
```

Tämä vaihtaa vuosinumeroiden käsittelyn kaikkialla:

- Luokan rajapinnassa kaikki vuosinumerot muuttuvat etumerkillisiksi.
- Mikäli luokan käyttäjät ovat käyttäneet tyyppiä `Paivays::VuosiNro` omassa koodissaan, heidänkin koodinsa muuttuu muutoksen mukaiseksi.
- Mikäli luokan toteutuksessa on käytetty `VuosiNro`-tyyppiä, parhaassa tapauksessa toteutuksen koodiin ei tarvitse koskea ollenkaan, koska muutos tapahtuu automaattisesti.

Luokan sisällä määriteltyihin tyyppeihin liittyy yksi C++:n omituisuus, joka on hyvä tietää. Luokan jäsenfunktioiden toteutuksissa (siis itse koodissa) kääntäjä tietää vasta jäsenfunktion nimen luettuaan, minkä luokan jäsenfunktiosta on kysymys. Niinpä kääntäjä ei jäsenfunktion paluutyyppiä lukiessaan vielä tiedä, mitä luokkaa ollaan käsittelemässä, eikä se näin ollen myöskään osaa automaattisesti hyväksyä luokan määrittelemiä tyyppejä. Tämän vuoksi täytyy jäsenfunktioiden toteutuksissa paluutyypit kirjoittaa muodossa `Luokkanimi::Tyyppi`, jos ne ovat luokan sisäisiä tyyppejä. Sen sijaan jäsenfunktioiden parametrilistassa ja itse koodissa voi luokan omia tyyppejä käyttää suoraan ilman luokan nimeä. Listaus 8.6 näyttää esimerkkinä luokan `Paivays` jäsenfunktion `annaPaiva` toteutuksen.

8.4 Ohjelmakomponentin sisäiset rajapinnat

Joskus ohjelmiston rakenteesta tulee kaikesta yrittämisestä huolimatta sellainen, että kaikkea luokan sisäiseen toteutukseen liittyvää toiminnallisuutta ei saada luokan sisään kapseloiduksi, vaan osa siitä joudutaan — usein ohjelmointikielen syntaksista johtuvista syistä — toteuttamaan luokan ulkopuolisissa funktioissa. Tällöin ongelmaksi tulee, että vain luokan omilla funktioilla on pääsy luokan sisäiseen

```
1 Paivays::PaivaNro Paivays::annaPaiva() const
2 {
3   return paiva_;
4 }
```

LISTAUS 8.6: Luokan määrittelemä tyyppi paluutyyppinä

toteutukseen, joten luokan ulkopuolinen funktio joutuu periaatteessa käyttämään luokkaa sen julkisen rajapinnan kautta.

Yleensä tilanne on sellainen, että vaikka ongelmallinen funktio onkin luokan ulkopuolinen, se kuitenkin kuuluu käsitteellisesti samaan moduuliin tai komponenttiin kuin luokkakin. Täten ongelmana ei niinkään ole se, että ohjelmassa haluttaisiin rikkoa luokan tarjoamaa kapselointia vaan se, että ohjelmassa yhdelle funktiolle haluttaisiin sallia muuta ohjelmaa laajempi rajapinta luokan olioiden käyttämiseen. Vastaavasti komponentin sisällä olevilla luokilla saattaa olla tarve käyttää toisiaan tavoilla, joita ei haluta sallia komponentin ulkopuolella. Komponentin luokat tarvitsisivat tällöin toisiinsa normaalia julkista rajapintaa laajemman rajapinnan.

Mekanismit tällaisille komponentin sisäisille rajapinnoille vaihtelevat suuresti ohjelmointikielestä toiseen. Joissain ohjelmointikielissä (kuten C) tukea ei ole juuri lainkaan, jolloin ohjelmoija joutuu turvautumaan dokumentointiin, sopimuksiin ja koodauskäytäntöihin tarjotakseen moduulin sisälle ulkomaailmalle näkyvää laajemman rajapinnan.

Sen sijaan Java tarjoaa **pakkauksen** (*package*)käsitteen, joka on tarkoitettu moduulien kirjoittamiseen (tätä käsiteltiin lyhyesti aliluvussa 1.6.2). Javan luokissa on puolestaan rajapintojen **`public`**, **`protected`** ja **`private`** lisäksi mahdollisuus "pakkauksen sisäiseen" rajapintaan, joka saadaan aikaan kirjoittamalla jäsenfunktio tai -muuttuja ilman näkyvyysmäärettä. Näihin jäsenfunktioihin ja -muuttujiin pääsee käsiksi myös muista saman pakkauksen luokista.

C++:ssa vastaava rakenne saadaan aikaan **ystäväfunktioiden** (*friend function*) ja **ystäväluokkien** (*friend class*) avulla. Niitä käsitellään tarkemmin seuraavissa aliluvuissa.

8.4.1 C++: Ystäväfunktiot

C++ tarjoaa varsin karkean mekanismin laajemman rajapinnan tarjoamiseen: luokka voi julistaa joukon funktioita "ystävikseen". Luokan ystäväfunktioiden koodi pääsee käsiksi myös luokan olioiden private-osiin, eli käytännössä sillä on samat oikeudet luokan olioihin kuin

luokan omilla jäsenfunktioilla. Esittelyn syntaksi on

```
class Luokka
{
   ⋮
  friend paluutyyppi funktionnimi(parametrit);
};
```

On huomattava, että kyseinen esittely *ei* tee ystäväfunktiosta luokan jäsenfunktiota vaan kyseessä on täysin erillinen normaali funktio, jolle vain sallitaan pääsy luokan olioiden private-osaan.

Listauksessa 8.7 seuraavalla sivulla on esimerkki luokasta, jonka rakentaja on private-puolella. Tämä tarkoittaa sitä, että luokan normaali käyttäjä ei pääse ollenkaan luomaan luokan olioita. Sen sijaan luokka määrää funktion `luoLukko` ystäväkseen, jolloin funktio voi luoda olioita ja palauttaa niitä osoittimen päässä paluuarvonaan. Näin `luoLukko`-funktiolla on käytössään laajempi rajapinta kuin tavallisella käyttäjällä.

Vaikka ystävämekanismi salliikin periaatteessa ystäväfunktioille oikeuden käpälöidä olioiden private-osia aivan miten tahansa, kannattaa kuitenkin muistaa kapselointiperiaate. Yleensä koodin selkeyttä parantaa, jos ystäväfunktiotkaan eivät käytä suoraan luokan sisäistä toteutusta (jäsenmuuttujia), vaan luokan private-puolelle kirjoitetaan sopivat jäsenfunktiot, joiden kautta ystäväfunktio saa käyttöönsä laajemmat oikeudet. Tällä tavoin luokan rajapinta säilyy edelleen funktioista koostuvana.

8.4.2 C++: Ystäväluokat

Ystäväfunktioiden avulla voidaan yhdelle funktiolle sallia normaalia vapaampi pääsy tietyn luokan olioon. Jos ohjelmakomponentissa on kaksi toisiinsa kiinteästi liittyvää luokkaa, voi käydä niin, että luokassa on useita jäsenfunktioita, joiden täytyy saada vapaampi pääsy toisen luokan sisälle. Tämä olisi mahdollista toteuttaa luettelemalla kaikki luokan jäsenfunktiot toisen luokan ystäväfunktioiksi, mutta tämä on kömpelöä. C++ antaa myös luokalle mahdollisuuden julistaa toinen *luokka* ystäväkseen. Tämä tarkoittaa sitä, että kaikki kyseisen luokan jäsenfunktiot ovat automaattisesti ystäväfunktioita.

```
 1 class Lukko
 2 {
 3 public:
       ⋮
 4 private:
 5   Lukko(); // Rakentaja private-puolella: olioiden luominen mahdotonta!
 6   ~Lukko(); // Purkaja private-puolella: olioiden tuhoaminen mahdotonta!
       ⋮
 7   friend Lukko* luoLukko(); // luo Lukko-olioita
 8   friend void poistaLukko(Lukko* lp); // tuhoaa Lukko-olioita
 9 };
10
11 Lukko* luoLukko()
12 {
13   Lukko* lp = new Lukko; // Mahdollista, koska ystävä
       ⋮
14   return lp;
15 }
16 void poistaLukko(Lukko* lp)
17 {
18   delete lp; lp = 0; // Mahdollista, koska ystävä
19 }
20
```

LISTAUS 8.7: Laajemman rajapinnan salliminen ystäväfunktioille

Luokkaystävyys saadaan aikaan määreellä **`friend class`** `Luokkanimi` sen luokan esittelyssä, joka haluaa sallia toisen luokan jäsenfunktioille vapaan pääsyn omien olioidensa sisälle. Listaus 8.8 seuraavalla sivulla näyttää esimerkin tällaisesta esittelystä.

```
1  class LainausJarjestelma
2  {
       ⋮
3  };
4
5  class KirjastonKirja
6  {
7  public:
       ⋮
8    Paivays const& annaPalautusPvm() const;
9  private:
10   void asetaPalautusPvm(Paivays const& uusiPvm);
11   Paivays palautusPvm_;
       ⋮
12   friend class LainausJarjestelma;
13   // LainausJarjestelman oliot kutsuvat funktiota asetaPalautusPvm
14 };
```

LISTAUS 8.8: Ystäväluokat

Luku 9

Geneerisyys

ISOC++ 14.7.3/7: The placement of explicit specialization declarations for function templates, class templates, member functions of class templates, static data members of class templates, member classes of class templates, member class templates of class templates, member function templates of class templates, member functions of member templates of class templates, member functions of member templates of non-template classes, member function templates of member classes of class templates, etc., and the placement of partial specialization declarations of class templates, member class templates of non-template classes, member class templates of class templates, etc., can affect whether a program is well-formed according to the relative positioning of the explicit specialization declarations and their points of instantiation in the translation unit as specified above and below.
When writing a specialization,
be careful about its location;
or to make it compile
will be such a trial
as to kindle its self-immolation.

– International Standard 14882 [ISO, 1998]

Mallit kertovat meille, millainen jonkin asian pitäisi olla, miltä jonkin tulisi näyttää tai miten jokin saadaan rakennetuksi. Jokaisella suomalaisella on käsitys siitä, mikä sauna on. Saunan mallinen rakennus sisältää löylytilan, pukuhuoneen ja vilvoittelualueen mielellään veden äärellä. Saunojen rakentamiseen on paljon perinnetietoa, ohjeita

ja jopa tutkimustuloksia. Voidaan sanoa, että suomalaisten saunojen rakentaminen on dokumentoitu (mallinnettu) huolellisesti vuosisatojen saatossa.

Vaikka ohjelmistojen tekeminen on hyvin nuori ala saunojen rakentamiseen verrattuna, tälläkin alalla pyritään hyödyntämään aikaisempia kokemuksia. Ohjelmistomallit pyrkivät kertomaan yleisiä linjoja (geneerisen mallin) siitä, minkälainen ratkaisu parhaiten (kokemukseen perustuen) sopisi käsillä olevaan ongelmaan.

9.1 Yleiskäyttöisyys, pysyvyys ja vaihtelevuus

Uudelleenkäyttöä mainostetaan yhtenä olio-ohjelmoinnin suurena etuna. Sen saavuttaminen ei kuitenkaan ole niin helppoa kuin pintapuolisesti voisi luulla. Kun luokkaa tai ohjelmakomponenttia suunnitellaan, se räätälöidään ja kehitetään usein enemmän tai vähemmän tietoisesti juuri tiettyyn käyttöympäristöön. Tällöin on todennäköistä, ettei tuo komponentti kuitenkaan sovi täsmälleen samanlaisena toiseen käyttötarkoitukseen — uudelleenkäyttö ei onnistukaan puhtaassa muodossaan.

Syynä tähän on, että vaikka yleisellä tasolla jokin luokka tai komponentti vaikuttaisikin "uudelleenkäytettävältä", on sen käyttökohteilla kuitenkin hieman erilaisia tarpeita, jotka sitten vaikuttavat luokan rajapintaan tai toteutukseen. Tämän vuoksi ollaan yhä enemmän sitä mieltä, että uudelleenkäyttöä on erittäin vaikea saada aikaan "sattumalta". Uudelleenkäyttö vaatii sen, että luokkaa tai komponenttia ensimmäistä kertaa suunniteltaessa on edes jonkinlainen käsitys siitä, missä kaikissa ympäristöissä luokkaa tai moduulia tarvitaan ja käytetään. Tässä mielessä olisikin ehkä paras puhua **yleiskäyttöisyydestä** (*genericity*) mielummin kuin uudelleenkäytöstä (re-usability).

Yleiskäyttöisyyden suunnittelussa oleellinen asia on löytää komponentin mahdolliset käyttökohteet (tai edes arvata "todennäköisesti kattava" osa niistä) ja analysoida näiden käyttökohteiden tarpeita. Tällöin saadaan kuva siitä, missä osissa komponenttia eri käyttökohteiden tarpeet pysyvät täsmälleen samoina ja missä asioissa ne vaihtelevat. Tällainen **pysyvyys- ja vaihtelevuusanalyysi** (*commonality and variability analysis*) on tärkeä osa yleiskäyttöisen komponentin suunnittelua, koska se kertoo, missä määrin mahdollisuutta

yleiskäyttöisyyteen sovellusalueessa on ja missä osissa komponenttia se ilmenee. [Coplien, 1999]

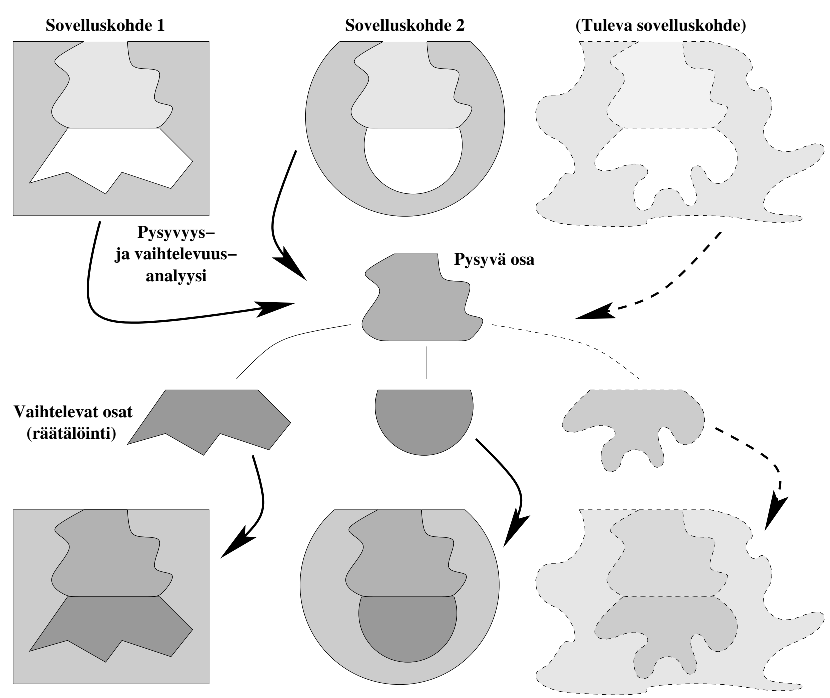
Äärimmäisessä tapauksessa pysyvyys- ja vaihtelevuusanalyysi saattaa antaa tulokseksi, että kaikkien käyttökohteiden tarpeet suunniteltavan komponentin kannalta ovat täsmälleen samat. Tällöin yleiskäyttöisyys on erittäin helppoa saada aikaan, koska *täsmälleen samanlainen* komponentti kelpaa kaikkialle, ja tehtäväksi jää enää komponentin toteuttaminen perinteisin keinoin. Tämä on tietysti ihannetilanne, joka esiintyy varsin harvoin. Yleensä käyttökohteiden tarpeissa on jonkinlaisia eroja, ainakin jos suunniteltava komponentti ei ole todella pieni.

Toisessa ääripäässä saatetaan huomata, että vaikka aluksi eri käyttökohteet näyttivätkin tarvitsevan samanlaista ohjelmakomponenttia, ovat niiden tarpeet niin erilaiset, ettei täydellisiä yhtäläisyyksiä käyttökohteiden välillä kuitenkaan ole. Tällaisessa tapauksessa vaihtelevuus on siis niin suurta, että käytännössä suunniteltava komponentti jouduttaisiin räätälöimään kokonaan jokaista käyttökohdetta kohti. Tällöin yleiskäyttöisyyttä ei tietenkään voida saavuttaa ollenkaan. Onneksi tällaiset tapaukset ovat myös erittäin harvinaisia.

Tyypillinen tulos pysyvyys- ja vaihtelevuusanalyysistä on, että *osa* komponentista voidaan pitää täsmälleen samanlaisena, osa taas täytyy toteuttaa eri tavalla eri käyttökohteisiin. Haasteena on toteuttaa yleiskäyttöinen komponentti niin, että pysyvät osat ja vaihtelevat osat saadaan kokonaan eriytettyä toisistaan.

Tällöin komponentin käyttäminen helpottuu, koska ohjelmoijalle on selvää, mitkä osat hänen täytyy räätälöidä tai kirjoittaa uudelleen juuri tiettyä käyttökohdetta varten, mitkä osat taas käytetään aina sellaisinaan. Mikäli analyysi on tehty kunnolla, on vielä lisäksi todennäköistä, että kun tulevaisuudessa käyttökohteita tulee lisää, pysyvät osat kelpaavat sellaisenaan niihinkin. Kuva 9.1 seuraavalla sivulla esittää pysyvyys- ja vaihtelevuusanalyysin toimintaa graafisesti.

Oman kiemuransa tällaiseen yleiskäyttöisen komponentin suunnitteluun tuo kaiken lisäksi se, että varsin usein “käyttökohteiden” valinta on varsin mielivaltaista. Tyypillisesti mitä suppeammaksi komponentin käyttökohdevalikoima määritellään, sitä enemmän pysyvyyttä komponenttiin saadaan, koska käyttökohteiden keskinäisiä eroja on vähemmän. Tällöin suuri osa komponentin koodista on täydellisesti uudelleenkäytettävää, mutta sen käyttöalue on suppea.



KUVA 9.1: Pysyvyys- ja vaihtelevuusanalyysi ja yleiskäyttöisyys

Toisaalta, jos käyttökohteiden kirjo on suuri, vaihtelevuuden määrä kasvaa suureksi. Tuloksena on komponentti, joka on kyllä todella yleiskäyttöinen, mutta jossa samanlaisena uudelleenkäytettävän koodin määrä on kuitenkin vähäinen. Kuten niin usein ohjelmistotekniikassa tässäkin taitoa vaatii sopivan kompromissin löytäminen, jotta hyödyt saadaan maksimoitua.

Olennaiseksi kysymykseksi yleiskäyttöisyydessä muodostuu yleensä se, miten komponentin pysyvyys ja vaihtelevuus saadaan erotettua toisistaan selkeästi. Tähän on olemassa lukematon määrä erilaisia vaihtoehtoja tilanteesta riippuen. Niitä käsitellään tarkemmin esimerkiksi kirjassa “Generative Programming” [Czarnecki ja Eisenecker, 2000]. Aliluvussa 9.2 esiteltävät suunnittelumallit ovat

eräänlainen esimerkki suunnittelutason pysyvyyden ja vaihtelevuuden hallinnasta. Aliluvussa 9.4 kuvataan lyhyesti pysyvyyden ja vaihtelevuuden hallintaa periytymisen avulla. Aliluvussa 9.5 taas käsitellään **C++:n mallit** (*template*), jotka antavat mahdollisuuden periytymisestä poikkeavaan yleiskäyttöisyyteen.

9.2 Suunnittelun geneerisyys: suunnittelumallit

Olio-ohjelmassa useat oliot toteuttavat yhdessä ohjelmiston toiminnallisuuden. Hyvin suunniteltu luokka mahdollistaa sen toteuttaman rakenteen uudelleenkäytön myös muualla kuin alkuperäisessä käyttökohteessa. Vastaavasti voimme uudelleenkäyttää usean luokan muodostaman toiminnallisen kokonaisuuden ja jopa yleistää tämän uudelleenkäytön säännöstöksi, joka kertoo, miten jokin yleisempi ongelma voidaan ratkaista kyseisen oliojoukon avulla. Tällaisia säännöstöjä nimitetään olio-ohjelmoinnin **suunnittelumalleiksi** (*design pattern*).

Suunnittelumalleja voi esiintyä usealla tasolla. Yleisen tason malli (**arkkitehtuurimalli**) voi kertoa periaatteen koko ohjelmiston rakenteesta (esimerkiksi WWW-sovelluksen osien jako http-palvelimen, "cgi-skriptien" ja tietokannan ohjauksen kesken). Keskitaso on moduulisuunnittelussa, jossa suunnittelumalli antaa mallin muutaman luokan (tyypillisesti 3–7) muodostamasta toiminnallisesta kokonaisuudesta. Lähinnä toteutusta ovat mallit (**toteutusmalli**, **idiomi**), jotka liittyvät tietyn ongelman ratkaisemiseen ohjelmointikielen tasolla (esimerkiksi funktionaalisten kielten rakenne `map`, joka kohdistaa määrätyn funktion kutsun yksitellen listan jokaiselle alkiolle) [Haikala ja Märijärvi, 2002]. Puhuttaessa suunnittelumalleista ilman lisämääreitä tarkoitetaan yleensä keskitason oliosuunnitteluun liittyviä malleja.

9.2.1 Suunnittelumallien edut ja haitat

Edellä mainitusta seuraa helposti, että suunnittelumalliksi voidaan julistaa melkein mikä tahansa "ei-triviaali" ohjelmakoodin tai suunnitelman osa. Tämä ei kuitenkaan ole tarkoitus. Yleisiksi suunnittelumalleiksi on tarkoitus kerätä *käytännössä* usein käytettyjä malliratkaisuja. Ensimmäinen suunnittelumallien kokoelma oli niin sa-

nottu “Gang of Four” (GoF) -kirja [Gamma *ja muut*, 1995], johon on valittu ainoastaan sellaisia suunnittelumalleja, joille on löytynyt vähintään kaksi toisistaan riippumatonta käyttäjäkuntaa (ohjelmistotaloa tai -projektia). Tällaiset yleiset suunnittelumallit tulisi pikkuhiljaa saada osaksi jokaisen ohjelmistosuunnittelijan tietämystä — tällä hetkellä on vain hyvin vaikea arvioida, mitkä mallit ohjelmistosuunnittelijan yleissivistykseen tulisi kuulua.

Parhaimmillaan suunnittelumalli on abstrakti suunnitteludokumentti, joka voidaan ottaa käyttöön ohjelmiston suunnittelussa heti, kun suunnittelija tunnistaa ongelmasta kohdan, johon malli sopii. Yleistä tietämystä (osa ohjelmoijan yleisivistystä) olevien mallien lisäksi suunnittelumallit ovat erinomainen tapa kerätä talteen organisaatiossa olevaa sovellusaluekohtaista tietoa, joka näin säilyy, vaikka ihmiset vaihtuvatkin.

Jotta suunnittelumalleista olisi hyötyä, ne tulisi dokumentoida huolellisesti ja niiden tulisi olla helposti omaksuttavissa. Dokumentointiin on määrämuotoisten dokumentointipohjien lisäksi kehitetty muodollisempia menetelmiä. **Mallikieli** (*pattern language*) on nimitys kokoelmalle saman sovellusalueen toisiinsa liittyviä suunnittelumalleja ja tiedolle siitä, miten näitä malleja voidaan sovellusalueella hyödyntää ja yhdistellä.

Suunnittelumallit itsessään eivät auta mitään, elleivät niitä käyttävät suunnittelijat tiedä, mitä mikin malli tarkoittaa. Nykyisin uusien suunnittelumallien löytäminen ja julkaiseminen vaikuttaa hieman riistäytyneen käsistä, eikä kukaan yksittäinen ohjelmoija ehdi opetella kaikkia uusia malleja. Mallien muistamista ja opettelua haittaa myös usein niiden sidonnaisuus englannin kieleen — vaatii erinomaista kielitaitoa saada oikea mielleyhtymä, jos suunnittelumallin nimenä on esimerkiksi Memento [Gamma *ja muut*, 1995, s. 283–291] tai The Percolation Pattern [Binder, 1999].

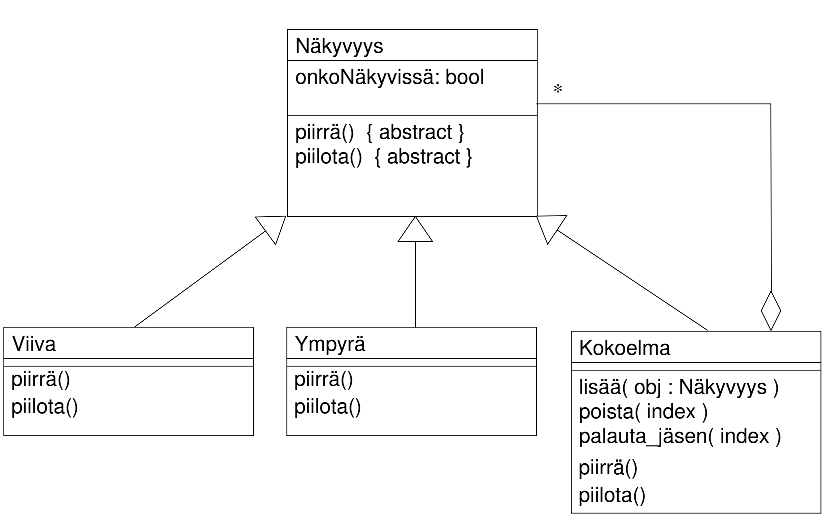
9.2.2 Suunnittelumallin rakenne

Yksi olio-ohjelmien perustoimenpiteistä on käsitellä joukkoa olioita vaikkapa kutsumalla joukon jokaiselle oliolle jotain palvelurutiinia. Polymorfismin yhteydessä (aliluku 6.5.2) näimme esimerkin oliojoukon käsittelystä luokkahierarkian avulla siten, että käsittely tapahtui yhteisen kantaluokan kautta. Joukkojen käsittelyssä seuraava luonteva askel on tarve määritellä alijoukkoja. Näitä uusia *oliokokoelmia*

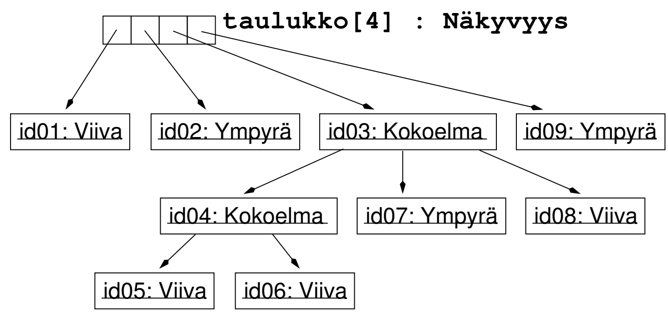
olisi luontevaa käsitellä samoissa paikoissa kuin kantaluokan olioita, koska muutoin olioita käsittelevän asiakasohjelmakoodin tulisi toimia aina eri tavoin sen mukaan, onko sillä käsittelyssä hierarkian perusolio vai kokoelmaolio. Ratkaisuna määrittelemme hierarkiaan uuden luokan Kokoelma, joka osaa käsitellä joukkoa alkuperäisiä olioita kantaluokan rajapinnan mukaisesti (katso kuva 9.2).

Kokoelman avulla voidaan luoda esimerkiksi kuvassa 9.3 seuraavalla sivulla esitetty rakenne. Kun luokan Kokoelma palvelu "piirrä" määritellään kutsumaan samaa palvelua jokaiselle kokoelmassa olevalle oliolle, saamme säilytetyksi myös alkuperäisen toiminnallisuuden, jossa taulukon kaikki grafiikkaoliot piirtyvät näyttölaitteelle käskettäessä. Nyt silmukka, joka kutsuu palvelua "piirrä" taulukon neljälle alkiolle, aiheuttaa palvelun kutsumisen kokoelman kautta kaikille rakenteessa oleville perusolioille (kuvassa id-tunnisteet 1, 2, 5, 6, 7, 8 ja 9).

Kuvan 9.2 rakenne on vasta tiettyyn tarkoitukseen laadittu suunnitelma. Siitä saadaan kuitenkin yleiskäyttöisempi, kun jätämme rakenteesta pois grafiikkaesimerkkimme ja keskitymme ainoastaan



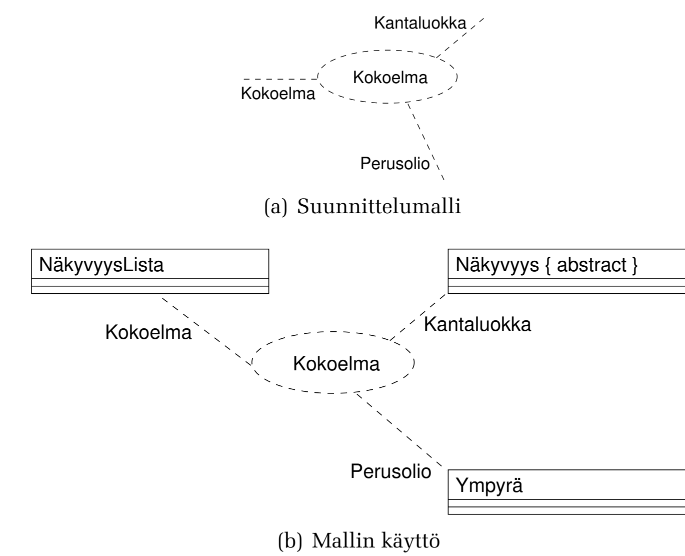
KUVA 9.2: Kokoelman toteuttava luokka



KUVA 9.3: Taulukko joka sisältää kokoelmia

olioiden kokoelmaan luokkahierarkiassa ja kantaluokan rajapinnan säilyttämiseen. Rakenteen tarkoituksena on tarjota saman kantaluokan kaksi eri variaatiota: määrätyillä operaatioilla toimivat oliot (kantaluokka on rajapintakuvaus) ja kokoelma näitä olioita. Rakenteen normaali käyttötarkoitus on ainoastaan rajapinnan käyttö ilman tietoa siitä, onko käytössä kokoelma vaiko perusolio. Näillä ehdoilla saamme kuvassa 9.5 sivulla 272 näkyvän rakenteen. Kyseessä on suunnittelumalli nimeltä **Kokoelma** (*Composite*) [Gamma *ja muut*, 1995, s. 163–173], joka on tarkemmin kuvattu aliluvussa 9.3.1.

UML määrittelee suunnittelumalleja varten oman piirrossymbolinsa, jossa kerrotaan ainoastaan mallin nimi (tämä erityisesti korostaa sitä, että suunnittelijan ja toteuttajan oletetaan tietävän heti mallin nimestä kaiken siihen liittyvän). Malliin liitetään tavallaan parametreina olioita, jotka toteuttavat suunnittelumallin määrittelemät osat. Näiden todellisten ohjelman olioiden sanotaan esiintyvän suunnittelumallin määrittelemässä **roolissa** (*role*). Esimerkki suunnittelumallin kuvaamisesta UML:llä on kuvassa 9.4 seuraavalla sivulla, jossa luokka Näkyvyys on roolissa *Kantaluokka*, Ympyrä on *Perusolio* ja NäkyvyysLista on *Kokoelma*.



(a) Suunnittelumalli

(b) Mallin käyttö

KUVA 9.4: Suunnittelumallin UML-symboli ja sen käyttö

9.3 Valikoituja esimerkkejä suunnittelumalleista

Tässä aliluvussa esitellään kolme melko yksinkertaista mutta hyvin yleisessä käytössä olevaa suunnittelumallia. Niistä näkyy GoF-kirjan ensimmäisenä käyttöön ottama suunnittelumallien dokumentointimuoto, jossa esitellään mallin käyttötarkoitus, perustelu sille miksi malli on yleiskäyttöinen ratkaisu useaan ongelmaan ja mallin luokkarakenteen kuvaus UML:n avulla.

9.3.1 Kokoelma (*Composite*)

Tarkoitus: Esittää olioita hierarkkisesti toisistaan koostuvina siten, että koosteolioita ja niiden osia voidaan käyttää samalla tavalla. [Gamma *ja muut*, 1995, s. 163–173]

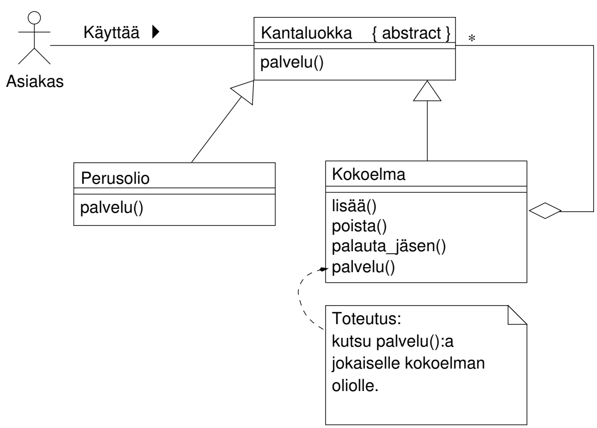
Perustelu: Oliokokoelmissa tarvitaan usein ominaisuutta tallettaa kokoelmaan toisia kokoelmia. Tällöin ongelmaksi tulee kokoelmien ja perusolioiden erilainen käyttäytyminen. Kun kokoelmaoliot ja perusoliot sisältävät saman toiminnallisen rajapinnan, tätä ongelmaa ei synny.

Soveltuvuus: Tarvitaan sisäkkäisiä kokoelmia tai halutaan, että olioita käsittelevien asiakkaiden ei tarvitse välittää perusolioiden ja kokoelmien välisistä eroista.

Rakenne: Katso kuva 9.5.

Osallistujat:

- **Asiakas:** Käyttää olioita kantaluokan tarjoaman rajapinnan kautta.
- **Kantaluokka:** Määrittelee (abstraktin) rajapinnan, jonka operaatiot aliluokkien on toteutettava.
- **Perusolio:** Primitiiviolio (joita voi olla useita), joka toteuttaa kantaluokan rajapinnan.



KUVA 9.5: Suunnittelumalli Kokoelma

- **Kokoelma:** Koosteluokka, joka tallettaa viittaukset osaolioihin. Kantaluokan rajapinta toteutetaan siten, että kukin kutsu välitetään jokaiselle koosteessa mukana olevalle oliolle.

Seuraukset: Asiakkaan ohjelmakoodi yksinkertaistuu, koska eroa perusolioiden ja koosteiden välille ei tarvitse tehdä. Uusien kooste- ja perusluokkien lisääminen on helppoa.

9.3.2 Iteraattori (*Iterator*)

Tarkoitus: Tarjoaa kokoelman alkioihin viittaamiseen rajapinnan, joka on erillään itse kokoelman toteutuksesta. [Gamma *ja muut*, 1995, s. 257–271]

Tunnetaan myös nimellä: Kohdistin (*Cursor*).

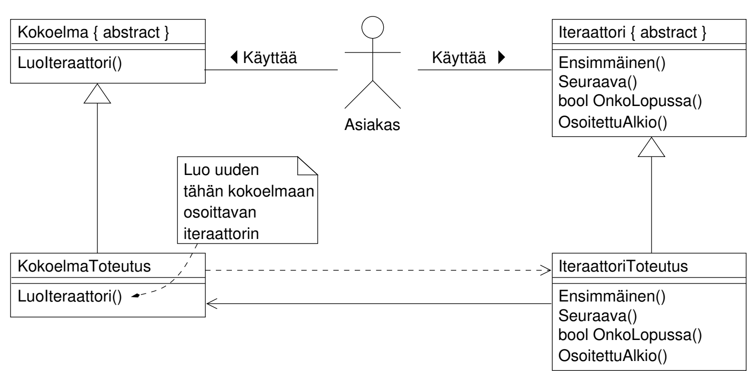
Perustelu: Listan tai muun alkiokokoelman rajapinta kasvaa helposti hyvin suureksi, jos siihen liitetään sekä rakenteen muokkaukseen että läpikäyntiin kuuluvat operaatiot. Yksi joukko läpikäynnin operaatioita (`alkuun`, `seuraava`, `edellinen` ynnä muut) mahdollistaa vain yhden läpikäynnin olemassaolon kerrallaan, koska alkiokokoelman toteuttava olio säilyttää itse operaatioiden vaatiman tilatiedon. Kun vastuu alkiokokoelman läpikäynnistä irrotetaan erilliseksi (mutta kokoelmaan läheisesti liittyväksi) olioksi, saadaan ratkaistuksi molemmat ongelmat.

Soveltuvuus: Tarvitaan useita yhtäaikaisia läpikäyntejä kokoelmaan tietoa tai halutaan tarjota yhtenäinen rajapinta useiden eri tavalla toteutettujen kokoelmien läpikäyntiin.

Rakenne: Katso kuva 9.6 seuraavalla sivulla.

Osallistujat:

- **Asiakas:** Käyttää rajapintaa Iteraattori luokan Kokoelma sisällä olevien alkioiden osoittamiseen ja läpikäyntiin.



KUVA 9.6: Suunnittelumalli Iteraattori

- **Kokoelma:** Määrittelee alkiokokoelman rajapinnan (erityisesti tavan, jolla kokoelmaan saadaan luoduksi iteraattori).
- **KokoelmaToteutus:** Toteuttaa edellä mainitun rajapinnan. Tarvitsee keinon luoda iteraattoriolion.
- **Iteraattori:** Määrittelee rajapinnan alkiokokoelman läpikäyntiin.
- **IteraattoriToteutus:** Toteuttaa edellä mainitun rajapinnan. (KokoelmaToteutus palauttaa tämän luokan instanssin.)

Seuraukset: Kokoelma pystyy tarjoamaan useita erilaisia tapoja alkioiden läpikäyntiin tarjoamalla useita iteraattoreita. Erillään oleva iteraattori yksinkertaistaa kokoelman rajapintaa. Kokoelman käyttäjä pystyy pitämään tarvittaessa useita osoituksia eli läpikäyntejä yhtäaikaa käynnissä.

9.3.3 Silta (*Bridge*)

Tarkoitus: Erottaa rajapinnan toteutuksesta siten, että toteutuksen muutokset eivät vaikuta mitenkään rajapinnan ohjelmakoodiin [Gamma *ja muut*, 1995, s. 151–161].

Tunnetaan myös nimellä: *Handle/Body*, *Envelope/Letter* [Coplien, 1992, luku 5.5] ja myös hieman omituisella nimellä *Cheshire Cat*[ϒ] [Meyers, 1998, s.148]. Lähes samasta rakenteesta on useita eri variaatioita, joita on esitelty artikkelissa "C++ Idioms Patterns" [Coplien, 2000].

Perustelu: Rajapinnan ja toteutuksen erotteleminen toisistaan ohjelmakoodin tasolla on joissain ohjelmointikielissä toteutettuna kielen rakenteilla (katso Modula-3, aliluku 1.6.1). C++:n luokkaesittelyssä on myös puhtaasti toteutukseen liittyviä osia (`private`-osuus), jolloin niissä tehdyt muutokset näkyvät myös julkisen rajapinnan käyttäjille (esimerkiksi käännösriippuvuuksina). Silta on toteutustekniikka, jolla myös C++:n tapaisissa kielissä saadaan rajapinta ja toteutus erotelluksi paremmin toisistaan (ortogonaalisempi yhteys osien välille).

Soveltuvuus: Halutaan poistaa kiinteä yhteys rajapinnan ja toteutuksen väliltä vaikkapa siksi, että toteutus voi muuttua ohjelman suorituksen aikana. Sekä rajapinnasta että toteutuksesta halutaan periyttää toisistaan riippumatta uusia versioita. Toteutuksen muutosten ei haluta aiheuttavan mitään muutoksia rajapintaa käyttävissä asiakaskoodeissa (niitä ei haluta kääntää uudelleen).

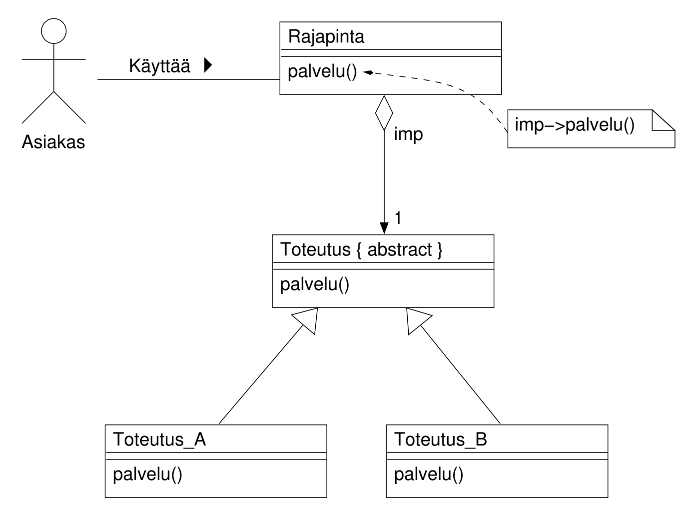
Rakenne: Katso kuva 9.7 seuraavalla sivulla.

Osallistujat:

- **Asiakas:** Käyttää palveluita luokasta Rajapinta tehdyn olion kautta.
- **Rajapinta:** Määrittelee julkisen rajapinnan ja viittaa olioon, joka toteuttaa tämän rajapinnan.
- **Toteutus:** Kantaluokka, josta periytetyt oliot ovat rajapinnan toteutusten eri versioita.

...

[ϒ] Viittaa *Lewis Carrollin* teoksessa "Alice's Adventures in Wonderland" [Carroll, 1865] olevaan kissaan, joka suomennoksessa "Liisa ihmemaassa" esiintyy nimellä Irvikissa. Tämä kissa pystyi erottamaan irvistyksensä muusta ruumiistaan niin, että vain irvistys näkyi.



KUVA 9.7: Suunnittelumalli Silta

Seuraukset: Toteutus ei ole pysyvästi kytketty rajapintaan, minkä vuoksi toteutus voi määräytyä ajoaikana ohjelman konfiguraatiosta tai jopa muuttua kesken ohjelman suorituksen. Rajapinnan ja toteutusten laajentaminen periyttämällä on yksinkertaisempaa. Rajapinnan käyttäjälle ei näy epäoleellista tietoa toteutusyksityiskohdista (C++).

9.3.4 C++: Esimerkki suunnittelumallin toteutuksesta

Tyypillinen esimerkki ajoaikaisesta toteutuksen valinnasta on nykyaikainen graafinen käyttöliittymä. Erityisesti UNIX-järjestelmissä käyttäjällä on valittavana useita erityylisiä käyttöliittymiä, jotka tarjoavat erilaisia vaihtoehtoja käytettävyydessä ja ulkoasussa. Jos haluamme tehdä ohjelman, joka tarjoaa todella joustavan käyttöliittymän, voimme valita käyttöön erilaisella käyttöliittymäkirjastolla tehdyn toteutuksen käyttäjän mieltymysten mukaan. Valinta voi olla etukäteen merkittynä ohjelman alustustiedoissa, jolloin käyttöliittymäkirjasto valitaan ohjelman käynnistyessä, tai erityisen joustava ohjelma voi pystyä käynnistämään koko käyttöliittymänsä uudelleen

ajoaikana tehdyn valinnan perusteella (joitain tällaisia järjestelmiä on olemassa). Tällaisessa tilanteessa voimme hyödyntää ohjelman rakenteessa suunnittelumallia silta (aliluku 9.3.3 sivulla 274).

Otamme esimerkiksi käyttöliittymän komponentin, jonka avulla voidaan näyttää käyttäjälle virhetiedotteita avaamalla näytölle ikkuna, jossa ilmoitusteksti sijaitsee. Tiedotteita käyttäville ohjelman osille tarjotaan abstrakti virheikkunan rajapinta, joka on esitetty listausessa 9.1 seuraavalla sivulla. Luokka `VirheIkkuna` on suunnittelumallin roolissa *Rajapinta*. Koska virheikkunan julkisen rajapinnan esittely ei tarvitse toteutuksen luokkaesittelyä, siitä on olemassa vain luokan ennakkoesittely (`VirheIkkunaToteutus`). Virheikkunan rajapinnan toteutuksessa ainoastaan **delegoidaan** eli välitetään rajapintakutsut varsinaiselle toteutusoliolle, joka on rajapinnan ainoan jäsenmuuttujan (`imp_`) päässä (tiedoston `virheikkuna.cc` rivit 11–14).

Toteutukselle määritellään kantaluokka (listaus 9.2 sivulla 279), josta periyttämällä tehdään todellisen toiminnallisuuden toteuttavia versioita (listaus 9.3 sivulla 279). Luokka `VirheIkkunaToteutus` on suunnittelumallin roolissa *Toteutus*. Nyt tämän luokkarakenteen avulla voidaan tehdä luokan `VirheIkkuna` instansseja erilaisilla toteutuksilla järjestelmän konfiguraation mukaan (listaus 9.4 sivulla 279).

9.4 Geneerisyys ja periytymisen rajoitukset

Aliluvussa 9.1 todettiin, että geneerisen ja yleiskäyttöisen komponentin suunnittelussa oleellinen asia on pysyvyyden ja vaihtelevuuden erottaminen toisistaan, jotta pysyvät osat voidaan toteuttaa vain kerran. Tämän tavoitteen saavuttamiseen löytyy ohjelmointikielitasolta erilaisia mekanismeja. Olio-ohjelmoinnissa periytyminen on mekanismi, jolla luokkien yhteiset ominaisuudet voidaan toteuttaa kertaalleen kantaluokassa. Niinpä onkin luonnollista että periytyminen antaa mahdollisuuden yleiskäyttöisyyteen. Tässä aliluvussa tutkitaan sitä, millaiseen yleiskäyttöisyyteen periytyminen antaa mahdollisuuden ja missä sen rajat tulevat vastaan.

On varsin helppoa keksiä tilanteita, joissa periytyminen on ihanteellinen mekanismi pysyvien ja vaihtelevien osien erottamiseen toisistaan. Jokainen periytymishierarkia vastaa tilannetta, jossa kantaluokkien toteutus periytyy aliluokille — kantaluokat ovat hierarkian

virheikkuna.hh

```
1  #ifndef VIRHEIKKUNA_HH
2  #define VIRHEIKKUNA_HH
3
4  #include <string>
5
6  class VirheIkkunaToteutus; // ennakkoesittely
7
8  class VirheIkkuna
9  {
10     public:
11         VirheIkkuna( VirheIkkunaToteutus const* p );
12         virtual ~VirheIkkuna();
13
14         // Näyttää viestin ikkunassa ja odottaa käyttäjän kuittausta
15         void NaytaIlmoitus( const std::string& viesti ) const;
16
17     private:
18         VirheIkkunaToteutus const* imp_; // osoitin toteutukseen
19 };
20 #endif
```

virheikkuna.cc

```
1  // Julkisen rajapinnan esittely
2  #include "virheikkuna.hh"
3
4  // Esittely rajapinnan toteutusten kantaluokasta
5  #include "virheikkunatoteutus.hh"
6
7  VirheIkkuna::VirheIkkuna( VirheIkkunaToteutus const* p ) : imp_(p) {};
8  VirheIkkuna::~VirheIkkuna() { imp_ = 0; }
9
10 // Palveluiden siirto toteutukselle:

11 void VirheIkkuna::NaytaIlmoitus( string const& viesti ) const
12 {
13     imp_->NaytaIlmoitus( viesti );
14 }
```

LISTAUS 9.1: Virhetiedoterajapinnan esittely ja toteutus

```
 1 #ifndef VIRHEIKKUNATOTEUTUS_HH
 2 #define VIRHEIKKUNATOTEUTUS_HH
 3
 4 #include <string>
 5
 6 class VirheIkkunaToteutus
 7 {
 8     public:
 9         VirheIkkunaToteutus();
10         virtual ~VirheIkkunaToteutus();
11
12         virtual void NaytaIlmoitus( const std::string& viesti ) const = 0;
13 };
14 #endif
```

Listaus 9.2: Toteutusten kantaluokka, `virheikkunatoteutus.hh`

```
1 // virheikkunaversiot.hh
2 #include "virheikkunatoteutus.hh"
3
4 class Gtk_Virheikkuna : public VirheIkkunaToteutus {
        ⋮
5
6 class Motif_Virheikkuna : public VirheIkkunaToteutus {
        ⋮
```

Listaus 9.3: Toteutus virheikkunoista, `virheikkunaversiot.hh`

```
 1 #include "virheikkuna.hh"
 2 #include "virheikkunaversiot.hh"
        ⋮
 3 VirheIkkuna* UusiVirheIkkuna()
 4 {
 5     VirheIkkunaToteutus* w;
 6     if( konfiguraatio.kaytossaMotif() ) {
 7         w = new Motif_VirheIkkuna();
 8     } else { // GTK
 9         w = new Gtk_VirheIkkuna();
10     }
11     return new VirheIkkuna( w );
12 }
```

Listaus 9.4: Eri virheikkunatoteutusten valinta

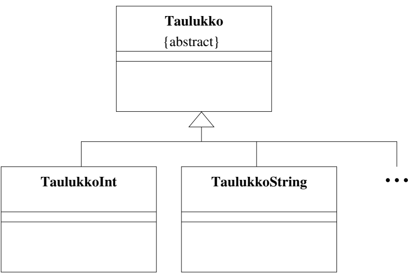
pysyvä osa, kun taas aliluokat voivat jokainen edustaa rajapinnaltaan tai toteutukseltaan erilaista "erikoistettua" versiota kantaluokan kuvaamasta asiasta. Tyypillisesti tällä tavalla toteutettu yleiskäyttöinen kirjasto tarjoaa valmiina luokkahierarkian yläosan (pysyvät osat), joista jokainen sovelluskohde sitten periyttää itselleen sopivat aliluokat, joissa toteutetaan vaihtelevat osat. Aliluvussa 6.11 mainitut sovelluskehykset ovat hyvä esimerkki periytymisen käyttämisestä tällä tavoin. Monissa oliokielissä periytyminen onkin pääasiallinen yleiskäyttöisyyden mekanismi.

Periytymisellä on kuitenkin muutama ominaisuus, joka rajoittaa sen käyttökelpoisuutta yleiskäyttöisyydessä. Tärkein niistä on se, että kaikki aliluokat perivät kantaluokan rajapinnan ja toteutukset *kokonaisuudessaan ja täsmälleen samanlaisena.* Tämä edellyttää, että yleiskäyttöisen komponentin pysyvyys koskee kokonaisia palveluita (jäsenfunktioita) parametreineen ja paluuarvoineen. Näin ei kuitenkaan aina ole.

Ongelmaa on ehkä helpoin kuvata esimerkin avulla. Oletetaan, että halutaan koodata periytymistä käyttäen yleiskäyttöinen taulukkotyyppi, suunnilleen samanlainen kuin `C++`:n `vector`-tyyppi. Tällaisessa taulukkotyypissä pysyvinä osina ovat itse vektorin toteutus ja käsittely, vaihtelevana osana puolestaan on taulukon alkioiden tyyppi, joka luonnollisesti riippuu käyttökohteesta. Periytymistä käyttäen tämä tarkoittaa, että suunnitellaan kaikille taulukoille yhteinen kantaluokka Taulukko, johon koodataan kaikille taulukoille yhteiset asiat. Tästä kantaluokasta sitten tarvittaessa periytetään kaikki erilaiset taulukot omiksi aliluokikseen, joihin sitten koodataan taulukon alkiotyypistä riippuvat asiat. Kuva 9.8 seuraavalla sivulla näyttää näin syntyvän luokkahierarkian.

Seuraavaksi tähän luokkahierarkiaan pitäisi lisätä tarvittavat julkisen rajapinnan palvelut. Kantaluokkaan Taulukko kuuluvat *kaikille taulukoille täsmälleen samanlaiset* palvelut, kun taas toisistaan eroavat jäsenfunktiot toteutetaan kussakin aliluokassa. Yksinkertaisuuden vuoksi tässä keskitytään pelkästään palveluihin `annaKoko`, `lisaaLoppuun`, `poistaLopusta` ja `haeAlkio`. Nämä vastaavat `vector`-tyypin operaatioita `size`, `push_back`, `pop_back` ja `at`.

Näistä palveluista `annaKoko` ei selvästi millään tavalla riipu alkioiden tyypistä, se kun vain palauttaa niiden lukumäärän. Sen voisi siis laittaa yhteiseen kantaluokkaan. Samalla tavoin `poistaLopusta` poistaa taulukon viimeisen alkion tyypistä riippumatta, joten sen voisi ai-



KUVA 9.8: Yleiskäyttöisen taulukon toteutus periyttämällä

nakin rajapintansa puolesta määritellä kantaluokassa. Sen sijaan palvelu `lisaaLoppuun` ottaa parametrikseen lisättävän alkion, joten palvelun parametrin tyyppi riippuu alkion tyypistä. Vastaavasti palvelu `haeAlkio` palauttaa paluuarvonaan halutun alkion, joten sen paluuarvon tyyppi riippuu alkiotyypistä. Nämä palvelut (ja kaikki muut vastaavat palvelut) pitäisi toteuttaa erikseen jokaisessa aliluokassa, jotta rajapinnan tyypit voivat erota toisistaan.

Kun tätä jaottelua jatketaan, huomataan että varsin suuressa osassa taulukon rajapintaa esiintyy alkion tyyppi jossain muodossa, joten varsin suuri osa palveluista siirtyisi aliluokkiin ja kantaluokan Taulukko pysyvä osa jäisi varsin pieneksi. Yleiskäyttöisyyden kannalta tämä on tietysti huono asia, koska uuden taulukkotyypin luomisessa täytyisi määritellä suuri joukko alkiotyypistä riippuvia palveluja, ja valmiina periytyvän toiminnallisuuden määrä jäisi pieneksi.

Yksi ratkaisu tähän ongelmaan on lisätä itse sovellusalueen pysyvyyttä. Vaaditaan, että kaikkien mahdollisten alkiotyyppien on *periydyttävä yhteisestä kantaluokasta Alkio.* Tämä sinänsä mitättömältä tuntuva lisävaatimus muuttaa tilannetta palveluiden kannalta oleellisesti. Nyt palvelun `lisaaLoppuun` parametrin tyyppi voikin olla Alkio, jolloin sama palvelu voi ottaa vastaan *minkä tahansa* luokasta Alkio periytetyn alkion.

Samoin palvelun `haeAlkio` paluuarvon tyyppi voi olla Alkio, jolloin sama palvelu pystyy palauttamaan minkä tahansa tyyppisen alkion. Tällä tavoin ajateltuna *kaikki* (tai ainakin lähes kaikki) taulukon palvelut saadaan toteutettua täsmälleen samalla rajapinnalla, ja ne

voidaan toteuttaa yhteisessä kantaluokassa. Tällöin periytettyjä taulukkoluokkia ei edes tarvita, koska vaihtelevuutta ei enää ole jäljellä!

Tämän tyyppistä ratkaisua käytetään esimerkiksi Java-kielen alkuperäisissä säiliöluokissa. Javassa kaikki luokat ovat periytettyinä yhdessä suuressa luokkahierarkiassa, jonka huipulla on kaikkien luokkien kantaluokka `Object`. Näin on luonnollista, että Javan taulukkoluokassa alkioiden tyyppi on aina `Object`, joten sama taulukkotyyppi kelpaa minkä tahansa tyyppisten alkioiden tallettamiseen.[♉] Yleiskäyttöisyyden kannalta tällainen tilanne on hyvä, koska sama luokka kelpaa sellaisenaan kaikkiin taulukoihin.

Yhdessä yleiskäyttöisessä taulukkoluokassa on myös huonot puolensa. Taulukkoa luotaessa ei enää ilmoiteta, *minkä tyyppisiä alkioita* taulukkoon on tarkoitus tallettaa. Koska palvelun `lisaaLoppuun` parametriksi kelpaa mikä tahansa yhteisestä kantaluokasta periytetty olio, voidaan samaan taulukkoon tallettaa sekaisin erityyppisiä alkioita, vaikkapa kokonaislukuja, merkkijonoja ja päiväyksiä. Joskus tällainen heterogeeninen useantyyppisiä alkioita sisältävä taulukko on kätevä, mutta sillä on haittapuolensa.

Samoin kuin `lisaaLoppuun`-palvelun parametri, palvelun `haeAlkio` paluutyyppi on myös yhteistä kantaluokkatyyppiä. Tämä tarkoittaa, että kun taulukosta haetaan alkio, ei kääntäjä pysty paluutyypin perusteella sanomaan, *minkä tyyppinen alkio taulukosta saatiin,* koska paluuarvona on osoitin tai viite yhteiseen kantaluokkaan. Käytännössä taulukon käyttäjän täytyy itse tietää, minkä tyyppinen alkio taulukosta *olisi pitänyt tulla,* ja tehdä tarvittava tyyppimuunnos, jotta alkio saadaan oikeantyyppisen osoittimen tai viitteen päähän.

Yhteisestä kantaluokasta aiheutuvat ominaisuudet johtavat siihen, että kääntäjä ei pysty tekemään taulukon käytöstä tarvittavia tyyppitarkastuksia, vaan ne jäävät taulukon käyttäjän vastuulle. Javan tyyppimuunnoksissa kantaluokasta aliluokkaan tehdään tarkastus siitä, että kantaluokkaviitteen päässä on oikeantyyppinen olio. Samoin olisi mahdollista lisätä taulukkoluokkaan tarkastus siitä, että kaikki taulukkoon lisättävät alkiot ovat keskenään samaa tyyppiä.

Tällaiset tarkastukset ovat kuitenkin väistämättä *ajoaikaisia eivätkä käännösaikaisia.* Niinpä helposta yleiskäyttöisyydestä on jouduttu maksamaan se hinta, että entistä suurempi osa ohjelman virheistä huomataan vasta testausvaiheessa (jos silloinkaan) sen sijaan, että

[♉] Tarkalleen ottaen tämä ei ole aivan totta, koska Javan perustyypit kuten **`int`** eivät ole olioita eivätkä näin ollen `Object`-luokasta periyttämällä luotuja.

kääntäjä ilmoittaisi niistä jo ohjelmaa käännettäessä. Samoin taulukkotyypin käytöstä on tullut työläämpää tarvittavien tyyppimuunnosten takia.

On syytä korostaa, että edellä mainitut ongelmat tulevat esille etupäässä käännösaikaisesti tyypitetyssä kielessä kuten Javassa. Smalltalk-kielen taulukkotyyppi on samantapainen kuin Javan. Koska kielessä ei kuitenkaan ole ollenkaan käännösaikaista tyypitystä, ovat kaikki tyyppitarkastukset joka tapauksessa *aina* ajoaikaisia, joten Smalltalkissa taulukkoluokka ei millään lailla huononna kielen tyyppiturvallisuutta (tai sen puutetta).

Pohjimmiltaan koko ongelman syy on se, että vaikka erityyppisillä taulukoilla onkin todella paljon yhteistä, ei tämä pysyvyys muodosta yhtenäistä kokonaisuutta, jonka saisi luontevasti toteutettua kantaluokkana. Taulukon alkioiden tyyppi vaikuttaa taulukon rajapintaan ja toteutukseen kauttaaltaan, eikä sen vaikutusta pysty eliminoimaan periytymistä käyttäen ilman, että tyyppiturvallisuudesta ja käyttömukavuudesta joudutaan tinkimään.

Tällaisten ongelmien vuoksi pysyvyyden ja vaihtelevuuden hallintaan tarvitaan jokin toinenkin mekanismi, joka antaa perinteisestä periytymisestä poikkeavan tavan yleiskäyttöisyyden hallintaan. C++:ssa tämä mekanismi on **mallit** (*templates*), jota käsitellään seuraavassa aliluvussa. Enemmän tai vähemmän samantapaisia mekanismeja on joissain muissakin kielissä. Muun muassa Java-kieleen tuli template-mekanismia muistuttava *generics*-mekanismi version 5.0 myötä.

9.5 C++: Toteutuksen geneerisyys: mallit (*template*)

Suunnittelumallien tarkoituksena on tarjota yleisiä suunnitteluratkaisuja, joista ohjelmiston suunnittelija sitten tuottaa oman suunnitelmansa korvaamalla mallissa esiintyvät roolit ja luokat oman ohjelmistonsa käsitteillä. Samaan tapaan myös ohjelmiston toteutusvaiheessa on usein tilanteita, joissa sama geneerinen koodirunko olisi käytännöllinen useassa kohtaa ohjelmaa.

Periaatteessa jo parametreja saavat funktiot ovat äärimmäisen yksinkertainen esimerkki tällaisesta geneerisyydestä. Funktioissa “auki jätettyjä” tietoja edustavat parametrit, joita funktion kirjoittaja käyttää

funktion koodissa varsinaisten arvojen sijaan. Funktiota kutsuttaessa näille parametreille sitten annetaan varsinaiset arvot.

Funktioiden tarjoama toteutuksen geneerisyys rajoittuu C++:ssa kuitenkin vain parametrien arvoihin. Kaikki muut asiat, kuten muuttujien tyypit ja taulukoiden koot, on kiinnitettävä jo koodausvaiheessa. Usein periaatteessa yleiskäyttöisessä koodissa kuitenkin myös muuttujien, parametrien yms. tyypit vaihtelevat. Esimerkiksi lukujen keskiarvo lasketaan samalla tavalla riippumatta siitä, ovatko luvut tyypiltään **`int`**, **`double`** vai kenties jokin itse luokkana määritelty lukutyyppi. Samoin itse tehty taulukkotyyppi on rakenteeltaan samanlainen riippumatta siitä, millaisia otuksia taulukkoon on tarkoitus tallettaa.

Niissä ohjelmointikielissä, joissa ei ole vahvaa (käännösaikaista) tyypitystä, ei ole myöskään näitä ongelmia. Ilman tyypitystä sama ohjelmakoodi pystyy laskemaan keskiarvon minkä tahansa tyyppisille arvoille ja taulukossa voi olla mitä asioita tahansa. Samalla tietysti jäädään ilman käännösaikaisia tyyppitarkastuksia ja mahdolliset virheet havaitaan vasta ajoaikana. Esimerkkejä tällaisista kielistä ovat Smalltalk, Python ja Scheme.

9.5.1 Mallit ja tyyppiparametrit

Käännösaikaisesti tyypitetyissä kielissä, kuten C++:ssa, täytyy kaikkien parametrien ja muuttujien tyyppien olla selvillä jo ohjelmaa käännettäessä, joten tyyppien geneerisyys ei onnistu ilman lisäkikkoja. **Mallit** (*template*) ovat C++:n tapa kirjoittaa yleiskäyttöisiä funktioita (**funktiomallit**, *function template*) ja luokkia (**luokkamallit**, *class template*). Malleissa tyyppejä ja joitain muitakin asioita voidaan jättää määräämättä (tästä tarkemmin aliluvussa 9.5.5). Auki jätetyt asiat "sidotaan" vasta myöhemmin käytön yhteydessä. Tällä tavoin saadaan säilytetyksi C++:n vahva käännösaikainen tyypitys, mutta annetaan silti mahdollisuus kirjoittaa geneeristä koodia.

C++:n kirjaston `vector` on esimerkki luokkamallista. C++:ssa jokaisen taulukon sisältämien alkioiden tyyppi täytyy tietää jo käännösaikana. Kuitenkin jokaisen taulukon toteutus on alkioiden tyyppiä lukuunottamatta sama. Niinpä `vector` itsessään on vasta luokkamalli, joka kertoo taulukoiden yleisen toteutuksen ja jättää auki alkioiden tyypin. Vektoreita käytettäessä ohjelmoija sitten kiinnittää alkioiden tyypin ja luo kokonaislukuvektoreita (`vector<int>`), liukulukuvektoreita

(`vector<double>`) tai päiväysosoitinvektoreita (`vector<Paivays*>`). Auki jätetyt asiat selviävät jo käännösaikana, mutta kuitenkin vasta vektoreiden käyttötilanteessa, ei itse vektorin ohjelmakoodissa.

Funktioissa parametrien nimiä käytetään koodissa korvaamaan parametrien "puuttuvia" arvoja, jotka selviävät kutsun yhteydessä. Samalla tavoin malleissa auki jätetyt tyypit nimetään ja ohjelmakoodissa käytetään näitä nimiä auki jätettyjen tyyppien tilalla. Yhdessä mallissa voi auki jätettyjä tyyppejä olla useita, aivan samoin kuin funktio voi saada useita parametreja. Koska funktion parametrit ja mallin auki jätetyt tyypit muistuttavat suuresti toisiaan, puhutaankin usein mallien **tyyppiparametreista**.

Koska mallien koodissa on auki jätettyjä asioita, kääntäjä ei pysty mallin tavatessaan tuottamaan siitä koodia. Vasta kun kyseistä mallia *käytetään* ohjelmassa ja annetaan tyyppiparametreille arvot, kääntäjä kääntää mallista koodin, jossa mallin auki jätetyt tyypit on korvattu annetuilla tyypeillä. Tätä kutsutaan mallin **instantioimiseksi** (*instantiation*). Kääntäjä kääntää mallista oman versionsa *jokaista* sellaista käyttökertaa kohti, jossa tyyppiparametrien arvot ovat erilaiset. Näin mallien käyttäminen ei yleensä vähennä tuotetun konekoodin määrää, vaikka mallit tietysti vähentävätkin ohjelmoijan koodausurakkaa.

Mallien syntaksi C++:ssa on yksinkertainen:

```
template<typename tyyppiparam1, typename tyyppiparam2, ...>
// Tähän normaali funktion tai luokan määrittely
```

Malli alkaa avainsanalla **`template`**, jonka jälkeen nimetään kulmasulkeissa kaikki mallissa auki jätetyt tyyppiparametrit. Tämän jälkeen seuraa itse mallin koodi, joka funktiomallin tapauksessa on tavallinen funktion määrittely ja luokkamallin tapauksessa luokan. Koodissa tyyppiparametreja voi käyttää aivan kuin normaaleja C++:n tyyppejä. C++ sallii myös syntaksin, jossa avainsanan **`typename`** tilalla käytetään tyyppiparametrilistassa avainsanaa **`class`**. Tämä "vanha" syntaksi on täysin identtinen yllämainitun kanssa, mutta sitä tuskin kannattaa käyttää, koska auki jätetyt tyypit voivat malleissa aina olla mitä tahansa tyyppejä, eivät välttämättä luokkia.

Seuraavassa esitellään C++:n template-mekanismin perusperiaatteet ja niitä käytetään jonkin verran hyväksi mm. luvussa 10. Template-mekanismi on kuitenkin varsin monimutkainen ja -puolinen niin syntaksiltaan kuin käyttötavoiltaan ja rajoituksiltaankin. C++:n

geneerisyydestä ja geneerisestä ohjelmoinnista kiinnostuneen kannattaakin tutustua esimerkiksi kirjoihin "C++ Templates — The Complete Guide" [Vandevoorde ja Josuttis, 2003] ja "Modern C++ Design" [Alexandrescu, 2001].

9.5.2 Funktiomallit

Funktiomallit (*function template*) ovat geneerisiä malleja, joista kääntäjä voi generoida eri tyypeillä toimivia funktioita. Kaikilla mallista generoiduilla funktioilla on sama nimi, mutta ne toimivat yleensä erityyppisillä parametreilla.

Listauksessa 9.5 on funktiomallin `min` toteutus. Rivi 1 kertoo, että kyseessä on malli ja että siinä on yksi auki jätetty tyyppi, jota mallin koodissa kutsutaan nimellä `T`. Riveillä 2–12 on sitten funktion määrittely, jossa tyyppiparametria `T` käytetään kuin mitä tahansa tyyppiä kertomaan, että mallista luodut funktiot ottavat kaksi samantyyppistä parametria ja palauttavat vielä samaa tyyppiä olevan paluuarvon. Tyyppiä `T` käytetään myös funktion rungossa tuloksen sisältävän paikallisen muuttujan luomiseen.

Kun funktiomalli on määritelty, voidaan siitä luoda todellisia funktioita yksinkertaisesti kutsumalla niitä. Mikäli kaikki tyyppiparametrit esiintyvät funktiomallin parametrilistassa, kääntäjä osaa automaattisesti päätellä kutsusta tyyppiparametrien arvot. Esimerkiksi kutsusta `min(1,2)` kääntäjä päättelee, että `T`:n on oltava **`int`**. Sen jäl-

```
1  template <typename T> // Tai template <class T> (identtinen)
2  T min(T p1, T p2)
3  {
4    T tulos;
5    if (p1 < p2)
6    {
7      tulos = p1;
8    } else {
9      tulos = p2;
10   }
11   return tulos;
12 }
```

LISTAUS 9.5: Parametreista pienemmän palauttava funktiomalli

keen kääntäjä kääntää automaattisesti funktiomallista koodin, jossa T on korvattu tyypillä **int**, ja kutsuu tätä koodia. Vastaavasti kutsun min(2.3, 5.7) nähdessään kääntäjä tuottaa funktiomallista koodin, jossa T on **double** (liukulukuvakiot ovat C++:ssa tyyppiä **double**).

Vaikka jokaisesta funktiomallin kutsusta periaatteessa tuotetaankin oma koodinsa, nykyiset kääntäjät ovat niin älykkäitä, että ne osaavat samantyyppisissä kutsuissa käyttää yhteistä koodia. Jos kääntäjä on jo kutsun min(1,2) yhteydessä tuottanut funktiomallista **int**-tyyppiä käyttävän toteutuksen, se kutsuu tätä samaa toteutusta myöhemmin tavatessaan kutsun min(5,9). Näin mallien käyttö ei kasvata ohjelmakoodin kokoa tarpeettomasti.

Vaikka kääntäjä osaakin päätellä tyyppiparametreille arvot kutsun parametrien tyypeistä, voidaan kutsun yhteydessä myös antaa tyyppiparametreille eksplisiittiset arvot kirjoittamalla kyseiset arvot kulmasulkeissa mallin nimen perään. Esimerkiksi kutsu

```
float f = min<float>(3.2, 6);
```

käskee kääntäjää tuottamaan koodin, jossa T on **float**, vaikka parametrien tyypit ovatkin **double** ja **int**. Kääntäjä tuottaa mallista min-funktion, joka ottaa parametreikseen kaksi **float**-arvoa ja palauttaa myös **float**-arvon. Koska kutsussa annetut parametrit ovat eri tyyppiä, sovelletaan niihin implisiittisiä tyyppimuunnoksia aivan kuten normaalistikin funktioita kutsuttaessa. Tällainen tyyppiparametrien eksplisiittinen määrääminen on joskus hyödyllistä, kun halutaan pakottaa juuri tietynlainen toteutus mallista. Se on myös välttämätöntä, jos funktiomallissa on tyyppiparametreja, jotka eivät käy ilmi mallin parametrilistasta (tällöin kääntäjä ei tietenkään voi itse päätellä niiden arvoja).

9.5.3 Luokkamallit

Luokkamalli (*class template*) toimii mallina luokille, jotka ovat muuten samanlaisia, mutta joissa jotkin tyypit voivat erota toisistaan. Listaus 9.6 seuraavalla sivulla esittelee luokkamallin Pari, joka kuvaa mallin luokille, joihin voi tallettaa kaksi mielivaltaista tyyppiä olevaa alkiota ja joista nämä alkiot voi lukea kutsuilla annaEka ja annaToka. Mallissa on kaksi auki jätettyä tyyppiä T1 ja T2, jotka esitellään mallin alussa ja joita sitten käytetään itse luokan määrittelyssä.

```
1  template <typename T1, typename T2>
2  class Pari
3  {
4  public:
5    Pari(T1 eka, T2 toka);
6    T1 annaEka() const;
7    T2 annaToka() const;
        ⋮
11 private:
12   T1 eka_;
13   T2 toka_;
14 };
```

LISTAUS 9.6: Tietotyypin "pari" määrittelevä luokkamalli

Itse `Pari` ei vielä ole luokka, vaan vasta malli kokonaiselle "perheelle" luokkia. Luokkamallista saadaan instantioitua todellinen luokka määräämällä tyyppiparametreille arvot. Tämä tapahtuu luettelemalla tyyppiparametrien arvot kulmasulkeissa luokkamallin nimen jälkeen. Esimerkiksi rivi

```
Pari<int, double> p(1, 3.2);
```

tuottaa ensin luokkamallista luokan, jossa `T1` on **`int`** ja `T2` on vastaavasti **`double`**. Tämän jälkeen tästä luokasta luodaan olio `p`. Tämä olio sisältää sitten tyyppejä **`int`** ja **`double`** olevat jäsenmuuttujat, sen jäsenfunktion `annaEka` paluutyyppi on **`int`** ja niin edelleen.

Olioiden luomisen lisäksi luokkamallista voi instantioida luokan missä tahansa kohtaa ohjelmaa. Syntaksi `Pari<int, double>` toimii luokkamallista luodun luokan nimenä, ja sitä voi käyttää missä tahansa kuten tavallista luokan nimeä. Esimerkiksi funktioesittely

```
void f(Pari<int, int>& i, Pari<float, int*> d);
```

esittelee tavallisen funktion, joka ottaa parametreikseen viitteen kaksi kokonaislukua sisältävään pariin ja parin, johon on talletettu liukuluku ja osoitin kokonaislukuun.

Jokainen erilainen luokkamallista luotu luokka on oma erillinen luokkansa, eikä sillä ole mitään sukulaisuussuhdetta muihin samasta mallista luotuihin luokkiin, joissa tyyppiparametrit ovat erilaiset. Esimerkiksi edellä olleet luokat `Pari<int,int>` ja `Pari<float,int*>`

ovat täysin omia luokkiaan, eivätkä niistä luodut oliot ole keskenään vaihtokelpoisia. Näin täytyy tietysti ollakin, koska sekä luokkien rajapinta että sisäinen toteutus eroavat toisistaan tyyppien osalta.

Listaus 9.6 vasta esitteli luokan. Tämän lisäksi täytyy luokkamallille kirjoittaa vielä jäsenfunktioiden toteutukset. Tämä tapahtuu C++:ssa niin, että jäsenfunktiot kirjoitetaan ikään kuin omiksi malleikseen, ja jokaiseen jäsenfunktion toteutukseen tulee oma **`template`**-määreensä tyyppiparametreineen kaikkineen. Listauksessa 9.7 on esimerkkinä kaksi `Pari`-luokkamallin jäsenfunktion toteutusta. Ne alkavat samanlaisella **`template`**-määreellä kuin itse luokkamallin esittelykin. Lisäksi niiden koodissa on ennen näkyvyystarkenninta `::` "luokan nimenä" merkintä `Pari<T1, T2>`, jossa mallin tyyppiparametrit on "sijoitettu paikoilleen".

Luokkamallin jäsenfunktiot käyttäytyvät kuten funktiomallit myös siinä mielessä, että kääntäjä instantioi luokkamallista instantioidulle luokalle vain ne jäsenfunktiot, joita luokan olioille todella kutsutaan. Normaalin luokan tapauksessahan kääntäjä kääntää kaikki jäsenfunktiot riippumatta siitä, kutsutaanko niitä. Tässä suhteessa luokkamallit voivat jopa vähentää tuotetun konekoodin määrää, jos luokkamallissa on paljon jäsenfunktioita, joita ei käytetä.

C++:ssa on myös mahdollista kirjoittaa **jäsenfunktiomalleja** (*member function template*). Jäsenfunktiomalli muistuttaa muuten tavallista funktiomallia, mutta se on määritelty jonkin luokan jäsenfunktioksi. Niinpä siihen pätevät kaikki asiat, jotka on selitetty funktiomalleista aliluvussa 9.5.2. Kaikki kääntäjät eivät vielä (kesällä 2000) tue jäsenfunktiomalleja, mutta uusimmissa kääntäjäversioissa tämä ominaisuus jo yleensä toimii.

```
1  template <typename T1, typename T2>
2  Pari<T1, T2>::Pari(T1 eka, T2 toka) : eka_(eka), toka_(toka)
3  {
4  }
5
6  template <typename T1, typename T2>
7  T1 Pari<T1, T2>::annaEka() const
8  {
9    return eka_;
10 }
```

LISTAUS 9.7: Esimerkki luokkamallin `Pari` jäsenfunktioista

Mielenkiintoinen yhdistelmä syntyy, kun luokkamallille määritellään jäsenfunktiomalli. Tällöin siis jo itse luokkamallissa on tietty määrä auki jätettyjä tyyppejä ja sen jäsenfunktiomallissa on näiden tyyppien lisäksi vielä joukko omia auki jätettyjä tyyppejä, jotka määräytyvät jäsenfunktiomallin kutsun yhteydessä. Koska tässä yhdistelmässä on ikään kuin malli mallin sisällä, se on syntaksiltaan varsin erikoinen. Listaus 9.8 sisältää esimerkin tällaisesta luokkamallin jäsenfunktiomallista. Tämän jäsenfunktiomallin avulla mihin tahansa pariin voi "summata" minkä tahansa tyyppisen toisen parin, kunhan vain parien alkiot voi laskea yhteen keskenään.

9.5.4 Tyyppiparametreille asetetut vaatimukset

Mallin määrittelyssä tyyppiparametreille ei suoraan aseteta mitään vaatimuksia vaan auki jätetyt tyypit vain nimetään mallin alussa, ja sen jälkeen niitä käytetään itse mallin koodissa. On kuitenkin selvää, ettei mallille voi antaa tyyppiparametreiksi mitä tahansa. Eri tyyppien ja olioiden rajapinnat eroavat toisistaan, ja näin ollen myös niiden käyttö on erilaista.

```
 1 template <typename T1, typename T2>
 2 class Pari
 3 {
       ⋮
 9   template <typename T3, typename T4>
10   void summaa(Pari<T3, T4> const& toinenPari);
       ⋮
14 };
       ⋮
27 template <typename T1, typename T2>
28 template <typename T3, typename T4>
29 void Pari<T1, T2>::summaa(Pari<T3, T4> const& toinenPari)
30 {
31   eka_ += toinenPari.annaEka();
32   toka_ += toinenPari.annaToka();
33 }
```

LISTAUS 9.8: Luokkamallin sisällä oleva jäsenfunktiomalli

C++:n malleissa ei millään tavalla erikseen kerrota, millainen rajapinta tyyppiparametreihin sijoitettavilla todellisilla tyypeillä tulisi olla. Mallille annetuilta tyypeiltä vaaditaan vain, että mallin koodin on käännyttävä, kun tyyppiparametrien tilalle sijoitetaan instantioinnissa todelliset tyypit.

Esimerkiksi listauksen 9.5 funktiomalli `min` vertailee parametrejaan pienempi kuin -operaattorilla $<$. Näin ollen `min`-mallia voi käyttää vain tyypeille, joiden arvoja voi vertailla $<$-operaattorilla. Samoin jos mallin koodi kutsuu auki jätettyä tyyppiä olevalle oliolle jotain jäsenfunktiota, voi kyseistä mallia käyttää vain sellaisten tyyppien kanssa, joista kyseinen jäsenfunktio löytyy.

Vaikka edellämainittu periaate on sinänsä hyvin yksinkertainen, se aiheuttaa helposti ongelmia mallien kirjoittamisessa, koska tyyppiparametrien vaatimukset on piilotettu ja siroteltu mallin koodin sekaan. Tämän vuoksi **geneerisessä ohjelmoinnissa** (*generic programming*) pitäisi ottaa aina huomioon seuraavat seikat:

- Tyyppiparametreihin kohdistuvat vaatimukset tulisi aina dokumentoida ja kerätä yhteen paikkaan, jotta mallin käyttäjä pystyy helposti näkemään, millaisten tyyppien kanssa mallia voi käyttää.

- Jotta malli olisi mahdollisimman yleiskäyttöinen, sen suunnittelussa tulisi kiinnittää huomiota siihen, ettei malli vaadi tyyppiparametreiltaan yhtään enempää kuin on tarpeen.

Edellä mainituista varsinkin jälkimmäinen kohta vaatii mallin kirjoittajalta huolellisuutta ja taitoa. C++:ssa implisiittiset tyyppimuunnokset, arvonvälitys yms. aiheuttavat sen, että sinänsä viattomalta näyttävä koodi saattaa kulissien takana tehdä varsin monimutkaisia asioita. Tällöin mallin koodista tulee helposti sellaista, että se vaatii tyyppiparametreiltaan ominaisuuksia, joita ohjelmoija ei ole ollenkaan suunnitellut.

Esimerkiksi listauksen 9.5 sinänsä viattoman näköinen `min`-malli vaatii selvästi tyyppiparametriltaan pienemmyysvertailun. Sen lisäksi kuitenkin auki jätettyä tyyppiä `T` olevat parametrit välitetään funktion sisään normaalia arvonvälitystä käyttäen, joka vaatii kopiorakentajan käyttöä (aliluku 7.3). Samoin paluuarvon palauttaminen tehdään kopiorakentajalla. Koodissa on rivi `T tulos;`, joten tyypiltä `T` vaaditaan oletusrakentajaa. Kaiken kukkuraksi `tulos`-muuttujaan si-

joitetaan kahdessa kohtaa uusi arvo, joten malli vaatii sijoitusoperaattorin olemassaolon. Näiden kaikkien vaatiminen rajoittaa `min`-mallin käyttöä aika lailla, koska oletusrakentaja ja sijoitus ovat operaatioita, joita läheskään kaikille luokille ei haluta kirjoittaa.

Pienellä suunnittelulla `min`-malli saadaan huomattavasti käyttäjäystävällisemmäksi. Käyttämällä viitteenvälitystä arvonvälityksen sijaan päästään eroon kopiorakentajan vaatimisesta. Poistamalla sinänsä tarpeeton `tulos`-olio saadaan lisäksi oletusrakentajan ja sijoitusoperaattorin käyttö poistettua. Listaus 9.9 sisältää paremman version `min`-mallista. Sen koodi vaatii tyyppiparametrilta enää ainostaan pienempi kuin -operaattorin olemassaoloa.

9.5.5 Erilaiset mallien parametrit

Tähän mennessä mallien yhteydessä on käsitelty vain yksinkertaisia tyyppiparametreja. Mallien parametreissa on lisäksi joitain lisäominaisuuksia, jotka joskus helpottavat mallien käyttöä.

Mallien oletusparametrit

C++:ssa voidaan tavallisten funktioiden parametreille antaa oletusarvoja, jolloin funktiokutsun lopusta alkaen voi jättää parametreja antamatta. Esimerkiksi funktioesittely

```
void f(int i = 1, double j = 3.14};
```

```
1  template <typename T>
2  T const& min(T const& p1, T const& p2)
3  {
4    if (p1 < p2)
5    {
6      return p1;
7    }
8    else
9    {
10     return p2;
11   }
12 }
```

Listaus 9.9: Parempi versio listauksen 9.5 funktiomallista

kertoo, että funktiolla `f` on kaksi parametria, joista ensimmäisellä on oletusarvo `1` ja toisella `3.14`. Funktiota voi kutsua kolmella eri tavalla:

- Kutsussa `f(5, 0.0)` ei ole mitään ihmeellistä, ja `i` saa arvon `5` ja `j`:lle tulee arvo `0.0`.
- Kutsu `f(3)` aiheuttaa sen, että `i` saa arvon `3` ja `j`:lle käytetään oletusarvoa `3.14`.
- Kutsu `f()` antaa molemmille parametreille oletusarvon, eli `i` on `1` ja `j` on `3.14`.

Mallien tyyppiparametreille pätevät samat säännöt. Tyyppiparametrilla voi olla oletusarvo, jota käytetään jos tyyppiparametria ei instantioinnin yhteydessä anneta. Listaus 9.10 sisältää luokkamallin `Pari2`, jonka ensimmäinen tyyppiparametri on oletusarvoisesti **`int`** ja toisen tyyppiparametrin oletusarvo on sama kuin ensimmäinen parametri. Mallin voi nyt instantioida kolmella eri tavalla:

- Syntaksilla `Pari2<double, string>` kaikki toimii kuten ennenkin, eli tuloksena on liukuluku-merkkijono-pari.
- Syntaksi `Pari2<double>` tuottaa parin, jossa parin molemmat arvot ovat liukulukuja.
- `Pari2<>` tuottaa kokonaislukuparin.

```
1 template <typename T1 = int, typename T2 = T1>
2 class Pari2
3 {
4 public:
5   Pari2(T1 eka, T2 toka);
6   T1 annaEka() const;
7   T2 annaToka() const;
      ⋮
8 };
```

LISTAUS 9.10: Mallin oletusparametrit

Vakioparametrit

Malleissa voi tyyppien lisäksi jättää auki myös tiettyjä *käännösaikaisia* vakioita, jotka jäävät mallin parametreiksi tyyppiparametrien tapaan. Auki jätettäväksi kelpaavia vakioita ovat

- kokonaislukuvakiot ja luettelotyyppien arvot (**enum**)
- osoitin globaaliin olioon tai funktioon
- viite globaaliin olioon tai funktioon
- osoitin jäsenfunktioon tai -muuttujaan.

Listauksessa 9.11 on luokkamalli MJono. Siitä luoduilla merkkijonoluokilla on kiinteä maksimipituus, joka annetaan mallin parametrina. Mallille on lisäksi määrätty oletusarvoinen maksimipituus 80, jota käytetään, jos maksimipituutta ei erikseen anneta. Koska esimerkissä jokainen eri maksimipituudella luotu merkkijonoluokka on oma erillinen luokkansa, eripituiset merkkijono-oliot eivät kuulu samaan luokkaan eikä niitä näin ollen normaalisti voi esimerkiksi sijoittaa toisiinsa.

Malliparametrit

Viimeisenä mallissa voi jättää auki myös toisen mallin, joka annetaan vasta instantioinnin yhteydessä. Tämä mahdollisuus mallin **malliparametreihin** (*template template parameter*) on varsin harvoin normaalikoodissa tarvittava ominaisuus, mutta se tekee mahdolliseksi varsin monimutkaisenkin geneerisen ohjelmoinnin.

```
1  template <unsigned long SIZE = 80>
2  class MJono
3  {
4  public:
5    MJono(char const* arvo);
6    char const* annaArvo() const;
7  private:
8    char taulukko[SIZE+1];
9  };
10 MJono<12> s1("Tuli täyteen");
```

LISTAUS 9.11: Malli, jolla on vakioparametri

Malliparametreja ei juuri käsitellä tässä teoksessa, mutta listauksessa 9.12 on esimerkki funktiomallista, jossa on jätetty auki yksi kaksi tyyppiparametria saava malli sekä lisäksi yksi tavallinen tyyppiparametri. Mallin `summaa` avulla voi nyt summata mistä tahansa kaksiparametrisesta luokkamallista instantioituja olioita, kunhan luokkamallin tyyppiparametrit ovat samat, ja se tarjoaa operaatiot `annaEka` ja `annaToka`.

9.5.6 Mallien erikoistus

Mallit ovat varsin tehokas tapa kirjoittaa yleistä koodia, joka on tarkoitettu toimimaan tyypeistä riippumatta. Joskus tulee kuitenkin vastaan tilanne, jossa juuri tietyn tyypin tapauksessa mallin koodi tulisikin toteuttaa eri tavalla. Syynä tähän saattaa olla tehokkuus- tai tilaoptimointi, tai kenties kyseinen tyyppi eroaa jollain olennaisella tavalla muista tyypeistä.

Mallien erikoistus (*template specialization*) antaa mahdollisuuden tällaisiin erikoistapauksiin. Kun malli antaa jollekin asialle yleisen toteutuksen, erikoistukset määrittelevät tähän poikkeuksia. Käyttäjän kannalta kaikki säilyy ennallaan, ja mallia voi käyttää normaalisti, mutta mallia instantioidessa kääntäjä saattaakin valita normaalin mallin koodin sijaan erikoistuksen tarjoaman koodin.

Luokkamallien yhteydessä listauksessa 9.6 ollut malli `Pari` on toteutettu siten, että sen molemmat alkiot on talletettu omiin jäsenmuuttujiinsa `eka_` ja `toka_`. Jos halutaan tehdä totuusarvopari `Pari<bool,bool>`, tämä toteutustapa tuhlaa muistia. Jokainen jäsen-

```
 1 template < template <typename T1, typename T2> class X, typename S>
 2 S summaa(X<S, S> const& x)
 3 {
 4   return x.annaEka() + x.annaToka();
 5 }
 6
 7 void kayta()
 8 {
 9   Pari<int, int> p(1, 2);
10   int tulos = summaa(p);
11 }
```

LISTAUS 9.12: Mallin malliparametri

muuttuja vie välttämättä vähintään yhden tavun muistia, vaikka periaatteessa kaksi totuusarvoa saisi helposti puristettua yhteenkin tavuun. Tämä tilaoptimointi voidaan toteuttaa `Pari`-mallin erikoistuksena.

Listaus 9.13 sisältää alkuperäisen mallin erikoistuksen, jossa on vain yksi jäsenmuuttuja `ekaJaToka_`, johon molemmat totuusarvot voidaan tallettaa C++:n bittioperaatioita käyttäen.

Luokkamallien erikoistuksen syntaksissa jätetään **`template`**-avainsanan jälkeen kulmasulkeet tyhjiksi (merkkinä siitä, että erikoistuksessa ei auki jätettyjä tyyppejä enää ole), ja ne tyypit, joita erikoistus koskee, merkitään mallin nimen jälkeen kulmasulkeisiin. Listauksessa on myös yhden jäsenfunktion erikoistuksen toteutus. Siinä **`template`**-avainsanaa ei tarvita ollenkaan vaan erikoistuksen tyypit merkitään suoraan luokan nimen yhteyteen.

Funktiomallin erikoistus on syntaksiltaan vastaava kuin luokkamallinkin. Siinä **`template`**-avainsanan jälkeiset kulmasulkeet ovat jälleen tyhjät, ja erikoistuksen kohteena olevat tyypit merkitään kulmasulkeisiin funktiomallin nimen jälkeen. Jos kaikki erikoistuksen tyyppiparametrien arvot voi päätellä parametrilistan avulla, funktion nimen jälkeisen tyyppilistan voi jättää halutessaan pois. Listaus 9.14 seuraavalla sivulla sisältää `min`-mallin erikoistuksen päiväysolioille.

Luokkamallien tapauksessa C++ sallii vielä **luokkamallin osittais-**

```
 1 template <>
 2 class Pari<bool, bool>
 3 {
 4 public:
 5    Pari(bool eka, bool toka);
 6    bool annaEka() const;
 7    bool annaToka() const;
 8    // summaa() puuttuu erikoistuksesta!
 9 private:
10    unsigned char ekaJaToka_; // Säästää muistia
11 };
12
13 bool Pari<bool, bool>::annaEka() const
14 {
15    return (ekaJaToka_ & 1) != 0;
16 }
```

LISTAUS 9.13: Luokkamallin `Pari` erikoistus totuusarvoille

```
31 template<>
32 Paivays const& min<Paivays>(Paivays const& p1, Paivays const& p2)
33 {
34   if (p1.paljonkoEdella(p2) > 0)
35   {
36     return p2;
37   } else {
38     return p1;
39   }
40 }
```

LISTAUS 9.14: Funktiomallin `min` erikoistus päiväyksille

erikoistuksen (*class template partial specialization*). Tässä varsin harvoin tarvitussa mekanismissa *osa* luokkamallin tyyppiparametreista sidotaan tiettyihin arvoihin mutta lopputuloksessa on vielä avoimia tyyppejä. Listaus 9.15 näyttää `Pari`-mallin osittaiserikoistuksen, jota käytetään, kun molemmat parin tyyppiparametrit ovat samat.

9.5.7 Mallien ongelmia ja ratkaisuja

C++:n mallien varsin omalaatuisen syntaksin lisäksi mallien suunnitteleminen on varsin vaativaa puuhaa. Geneerinen ohjelmointi itsessään on hankalaa, ja useimpien C++-kääntäjien lähes lukukelvottomat virheilmoitukset eivät mitenkään auta asiaa. Tämän lisäksi itse mal-

```
 1 template <typename T>
 2 class Pari<T, T>
 3 {
 4 public:
 5   Pari(T eka, T toka);
 6   T annaEka() const;
 7   T annaToka() const;
 8 private:
 9   T alkiot_[2];
10 };
11 template <typename T>
12 T Pari<T, T>::annaEka() const
13 { return alkiot_[0]; }
```

LISTAUS 9.15: Luokkamallin `Pari` osittaiserikoistus

lien koodaamisessa on tiettyjä C++-riippuvia asioita, jotka on hyvä tietää. Tähän alilukuun on kerätty joitain tällaisia seikkoja.

Mallin koodin sijoittelu ja export

Tavallisista funktioista poiketen kääntäjä kääntää mallin koodin vasta instantioinnin yhteydessä, eli kun mallia käytetään. Käytännössä tämä aiheuttaa vaatimuksen, että kääntäjällä täytyy olla tiedossaan mallin koodi silloin, kun mallia käyttävää koodia käännetään.

Normaalien funktioiden yhteydessä tätä rajoitusta ei ole vaan pelkkä funktion esittely riittää funktion kutsumiseen. Tämä tekee mahdolliseksi sen, että funktioista vain esittelyt laitetaan otsikkotiedostoihin, jotka sitten luetaan **#include**-komennolla jokaiseen tiedostoon, jossa funktioita kutsutaan. Funktioiden varsinainen määrittely (koodi) voi sitten olla omassa kooditiedostossaan, joka voidaan kääntää erikseen. Funktiota kutsuttaessa pelkkä funktion esittely riittää kääntäjälle funktiokutsun tekevän koodin tuottamiseen (tätä käsiteltiin jo aiemmin aliluvussa 1.4, katso kuva 1.6 sivulla 41).

Koska mallin koodin on oltava kääntäjän tiedossa mallia käytettäessä, normaalin mallin koodia ei voi jättää omaan erikseen käännettävään kooditiedostoonsa, vaan mallin koodi on luettava sisään joka paikassa jossa mallia käytetään. Käytännössä tämä tarkoittaa sitä, että yleensä mallin koko koodi kirjoitetaan otsikkotiedostoon.

Ohjelman modulaarisuuden kannalta mallin koodin sijoittaminen otsikkotiedostoon on huono asia. Modulaarisuuden perusperiaatteitahan on, että eri ohjelmamoduulien ei tarvitse tietää toistensa toteutusta, vaan pelkän rajapinnan (eli esittelyiden) lukeminen riittää. Tämän vuoksi C++:aan lisättiin standardoinnin yhteydessä avainsana **export**, jonka avulla mallien modulaarisuutta voidaan lisätä. Ikävä kyllä tämä avainsana on tätä kirjoitettaessa (keväällä 2003) toteutettu vain erittäin harvoissa kääntäjissä. Lisäksi käytännön kokemukset eräiden lehtiartikkeleiden [Sutter, 2002b] mukaan viittaavat siihen, että **export** ei kuitenkaan ratkaise kaikkia käännösriippuvuusongelmia toivotulla tavalla, vaikka se mahdollistaakin mallien esittelyn ja toteutuksen kirjoittamisen eri tiedostoihin.

Jos mallista annetaan vain esittely ja sen alussa ennen **template**-sanaa esiintyy avainsana **export**, tämä kertoo kääntäjälle että mallin koodi on muualla. Tällöin kääntäjä ei mallia käytettäessä vielä varsinaisesti instantioi mallia, vaan pistää ainoastaan muistiin, millaista

instantiointia tarvitaan. Mallin koodi kirjoitetaan nyt toiseen kooditiedostoon ja varustetaan myös avainsanalla **export**.

Tätä kooditiedostoa kääntäessään kääntäjä pistää muistiin sen, että mallin koodi löytyy tarvittaessa kyseisestä tiedostosta. Kun lopulta koko ohjelman objektitiedostoja linkitetään yhteen, linkkeri etsii tarvittavia mallien instansseja vastaavat mallien koodit sisältävät kooditiedostot ja kääntää niistä tarvittavan koodin.

Tällä tavoin mallit voidaan **export**-avainsanan avulla kirjoittaa samaan tapaan kuin muukin koodi ja laittaa otsikkotiedostoihin vain mallien esittelyt. Listauksessa 9.16 on esimerkki **export**-avainsanan käytöstä.

Tyyppi vai arvo — avainsana typename

Mallin koodissa ei auki jätetyistä tyypeistä tiedetä mitään. Normaalisti tämä ei haittaa, koska kääntäjä tuottaa mallista konekoodia vasta mallin käytön yhteydessä, jolloin tyyppiparametreja vastaavat tyypit-

```
.................................. max.hh ..................................
 1 export template <typename T>
 2 T const& max(T const& p1, T const& p2);
................................. main.cc .................................
 1 #include "max.hh"
 2 int main()
 3 {
 4   int m = max(4, 8);
        ⋮
 5 }
.................................. max.cc ..................................
 1 export template <typename T>
 2 T const& max(T const& p1, T const& p2)
 3 {
 4   if (p1 > p2)
 5   {
 6     return p1;
 7   } else {
 8     return p2;
 9   }
10 }
```

LISTAUS 9.16: Avainsanan **export** käyttö

kin ovat jo tiedossa. Tietyissä tilanteissa kääntäjälle täytyy kuitenkin kertoa enemmän tyyppiparametreista.

C++:ssa luokan sisällä voi määritellä jäsenfunktioita, jäsenmuuttujia ja luokan sisäisiä tyyppejä (aliluku 8.3). Ongelmaksi tulee, että jos `T` on luokka, voidaan syntaksilla `T::x` viitata jäsenmuuttujaan tai -funktioon nimeltä `x` tai luokan sisällä määriteltyyn tyyppiin nimeltä `x`. Jos nyt `T` on mallin tyyppiparametri, ei kääntäjällä ole mallin koodia lukiessaan mitään tapaa tietää, onko `x` jäsenfunktio tai -muuttuja vai tyyppi. Tämä tieto taas on tarpeen jo mallin koodin sisäänlukuvaiheessa, jotta kääntäjä ylipäätään pystyy ymmärtämään koodin ja selvittämään, onko siinä syntaksivirheitä.

Ongelma on ratkaistu C++:ssa määrittämällä, että edellä mainittua muotoa olevat luokan sisälle tapahtuvat viittaukset tulkitaan *aina* niin, että ne viittaavat joko jäsenfunktioon tai -muuttujaan. Jos mallin koodissa halutaan, että `T::x` onkin tyyppi, sen eteen täytyy kirjoittaa avainsana **`typename`**.

Listaus 9.17 valottaa tätä esimerkin avulla. Siinä funktiomalli `summaa` olettaa saavansa tyyppiparametrina tietorakenteen, jossa on kentät `eka` ja `toka` sekä lisäksi sisäinen tyyppi `arvotyyppi`, joka kertoo minkä tyyppisiä arvoja `eka` ja `toka` ovat. Funktiomalli haluaa palauttaa tämäntyyppisen arvon, joten sen paluutyypiksi on merkit-

```
1  template<typename T>
2  typename T::arvotyyppi summaa(T const& pari)
3  {
4    return pari.eka + pari.toka;
5  }
6
7  struct IntPari
8  {
9    typedef int arvotyyppi;
10   arvotyyppi eka;
11   arvotyyppi toka;
12 };
13
14 void kaytto(IntPari const& p)
15 {
16   int summa = summaa(p);
17 }
```

LISTAUS 9.17: Tyypin määrääminen avainsanalla **`typename`**

ty `T::arvotyyppi`. Oletusarvoisesti kääntäjä tulkitsee tämän `T`:n jäsenfunktioksi tai -muuttujaksi, joten merkinnän eteen täytyy lisätä **`typename`**. Listauksessa näkyy myös esimerkki tämän funktiomallin käytöstä.

Luku 10

Geneerinen ohjelmointi: STL ja metaohjelmointi

We define abstraction as selective ignorance — concentrating on the ideas that are relevant to the task at hand, and ignoring everything else — and we think that it is the most important idea in modern programming. The key to writing a successful program is knowing which parts of the problem to take into account, and which parts to ignore. Every programming language offers tools for creating useful abstractions, and every successful programmer knows how to use those tools.
– From the preface of Accelerated C++ [Koenig ja Moo, 2000]

Hyvä esimerkki mallien käytöstä ja geneerisestä ohjelmoinnista on C++-standardiin kuuluva kirjasto STL (*Standard Template Library*). STL määrittelee joukon tavallisimpia tietorakenteita ja niiden käyttöön tarkoitettuja algoritmeja. Tarkoituksena on, että ohjelmoijan ei tarvitsisi keksiä pyörää uudelleen ja kirjoittaa aina omia tietorakenteitaan vaan ohjelmissa voisi suoraan käyttää valmiiksi kirjoitettuja, tehokkaita ja optimoituja vakiokomponentteja.

STL perustuu lähes kokonaan malleihin. Se on toteutettavissa kokonaan C++:n vakio-ominaisuuksia käyttäen eikä vaadi kääntäjältä mitään "erityisominaisuuksia". Niinpä STL on hyvä esimerkki siitä, kuinka geneerisiä ratkaisuja C++:lla on mahdollista saada aikaan.

Usein geneerisessä ohjelmoinnissa tulee vastaan tilanteita, jossa geneerisen kirjaston olisi tarpeen mukauttaa omaa rakennettaan auki jätetyistä tyypeistä riippuen, valita parhaiten tilanteeseen sopivan algoritmi tai muuten toimia "älykkäästi". Tällaisesta päättelyjä käyttävästä geneerisyydestä, jossa geneerinen komponentti vaikuttaa omaan rakenteeseensa, käytetään yleensä nimitystä **metaohjelmointi** (*metaprogramming*) ja sitä käsitellään jonkin verran aliluvussa 10.6.

10.1 STL:n perusperiaatteita

Kaikkein tärkein geneerisyyden muoto STL:ssä on, että kaikissa sen tietorakenteissa alkioiden tyyppi on jätetty auki eli tietorakenteet on kirjoitettu ottamatta kantaa alkioiden tyyppiin. Alkioiden tyyppi määrätään vasta siinä vaiheessa, kun tietorakenteita varsinaisesti luodaan. Käytännössä tämä tarkoittaa, että kaikki STL:n tietorakenteet ovat luokkamalleja. Esimerkki tästä on `vector`, jossa `vector<int>` on kokonaislukuvektori, `vector<string>` merkkijonovektori ja niin edelleen.

STL:n toteutus mallien avulla antaa mahdollisuuden siihen, että vaikka itse STL:ssä alkioiden tyyppi on jätetty avoimeksi, kääntäjä voi kuitenkin tehdä STL:ää käyttävässä koodissa kaikki tarpeelliset tyyppitarkastukset ja antaa mahdolliset virheilmoitukset jo *käännösaikana*. Tässä suhteessa STL eroaa esimerkiksi Javan tietorakenteista. Niissä käytetään geneerisyyden aikaansaamiseksi periytymistä.

Kaikki Javan tietorakenteet sisältävät luokkaa `Object` olevia alkioita. Koska kaikki Javan luokat on periytetty tästä luokasta, voi tietorakenteeseen tallettaa mitä tahansa alkioita. Tämä kuitenkin tarkoittaa, että Javassa yhteen tietorakenteeseen voi sekoittaa minkä tahansa tyyppisiä alkioita. Kun alkiot myöhemmin luetaan ulos, viitteet niihin täytyy erikseen tyyppimuunnoksella muuntaa alkioiden todellista tyyppiä vastaaviksi. Jos alkion todellinen tyyppi onkin oletetusta poikkeava, tuloksena on *ajoaikainen* virheilmoitus. Tässä mielessä mallit ovat periytymistä turvallisempi keino tämän tyyppisen geneerisyyden toteuttamiseen.

10.1.1 STL:n rakenne

Alkioiden tyypin auki jättämisen lisäksi STL:ssä on paljon muutakin geneeristä. Tietorakenteita käyttävät algoritmit ovat tietorakenteista riippumattomia ja geneerisesti kirjoitettuja, joten niitä voi käyttää myös itse kirjoitettujen tietorakenteiden kanssa, kunhan vain tietyt vaatimukset täyttyvät. Samoin tietorakenteiden muistinhallinta (se, miten ja mistä tarvittava muisti varataan) on geneerisesti kirjoitettu, joten ohjelmoija voi itse vaikuttaa siihen. STL on myös hyvä esimerkki tiettyjen suunnittelumallien, erityisesti iteraattoreiden (aliluku 9.3.2) käytöstä.

Tämän luvun *ei* ole tarkoitus olla kattava STL-opas vaan tarkoituksena on kertoa STL:n perusperiaatteet ja käydä niitä läpi siinä laajuudessa, kun ne liittyvät olio-ohjelmointiin ja geneerisyyteen. STL:ssä on sellaisia hyödyllisiä ominaisuuksia ja yksityiskohtia, joiden läpikäymiseen ei tässä kirjassa ole mahdollisuuksia ja jotka eivät muutenkaan sovi hyvin tämän kirjan aihepiiriin. Tarkempaa tietoa STL:n yksityiskohdista ja sen käytöstä ohjelmointiin löytyy monista C++-oppikirjoista [Lippman ja Lajoie, 1997], [Stroustrup, 1997]. Lisäksi STL:n ja muun C++:n vakiokirjaston käytöstä löytyy kirja "The C++ Standard Library" [Josuttis, 1999].

STL muodostuu seuraavista osista:

- **Säiliöt** (*container*) ovat STL:n tarjoamia tietorakenteita. Niitä käsitellään aliluvussa 10.2.

- **Iteraattorit** (*iterator*) ovat "kirjanmerkkejä" säiliöiden läpikäymiseen. Aliluku 10.3 kertoo iteraattoreista tarkemmin.

- **Geneeriset algoritmit** (*generic algorithm*) käsittelevät säiliöitä iteraattoreiden avulla. Niistä kerrotaan aliluvussa 10.4.

- **Säiliösovittimet** (*container adaptor*) ovat säiliömalleja, jotka toteutetaan halutun toisen säiliön avulla. Niillä voi muuntaa säiliön rajapinnan toisenlaiseksi. Säiliösovittimia käsitellään lyhyesti aliluvun 10.2.3 lopussa.

- **Funktio-oliot** (*function object*) ovat olioita, jotka käyttäytyvät kuten funktiot ja joita voi käyttää muuan muassa algoritmien toiminnan säätämiseen. Niistä kerrotaan aliluvussa 10.5.

- **Varaimet** (*allocator*) ovat olioita säiliöiden muistinhallinnan räätälöintiin. Lyhyesti selitettynä varaimet ovat olioita, jotka osaavat varata ja vapauttaa muistia. Normaalisti STL:n säiliöt varaavat muistinsa **`new`**'llä ja vapauttavat **`delete`**llä. Jos niille annetaan ylimääräisenä tyyppiparametrina varainluokka, ne käyttävät ko. luokan palveluita tarvitsemansa muistin varaamiseen ja vapauttamiseen.

Nämä osat eivät muodosta irrallisia kokonaisuuksia, vaan käyttävät lähes kaikki toisiaan. Esimerkiksi säiliöitä voi tietyssä määrin käyttää ymmärtämättä muita STL:n osia, mutta algoritmien käyttö vaatii iteraattoreiden ymmärtämisen ja iteraattoreiden käyttö puolestaan säiliöiden. Funktio-olioita taas tarvitaan joidenkin algoritmien säätämiseen. STL:n ehkä vähiten käytetty osa on varaimet, joita tarvitaan vain jos halutaan pakottaa säiliöt varaamaan muistia juuri tietyllä tavallisuudesta poikkeavalla tavalla.

10.1.2 Algoritmien geneerisyys

Olio-ohjelmoinnissa on totuttu siihen, että luokan oliot sisältävät halutut tiedot ja tietoja käsittelevä koodi kirjoitetaan luokan jäsenfunktioihin. Tästä syystä tuntuisi alkuun loogiselta, että STL:n tarjoamat algoritmit olisi toteutettu säiliöiden jäsenfunktioina. Näin ei kuitenkaan ole tehty, vaan lähes kaikki STL:n algoritmit on kirjoitettu irrallisina funktioina, jotka eivät kuulu mihinkään luokkaan vaan ottavat kaiken tarvittavan tiedon parametreinaan. Mikä on syynä tähän oliovastaiseen toteutukseen?

Syy algoritmien toteuttamiseen tavallisina funktioina on, että STL:n algoritmien geneerisyys ei rajoitu käsiteltävien alkioiden tyypin auki jättämiseen. Kaikki algoritmit on toteutettu funktiomalleina. Samalla tavoin kuin säiliöt jättävät alkioidensa tyypin auki, STL:n algoritmit jättävät auki sen, *minkä tyyppisen tietorakenteen kanssa ne toimivat*. Näin samaa geneeristä `find`-algoritmia voidaan käyttää etsimään halutunlaista alkiota niin listasta, vektorista kuin joukostakin. Etuna tästä on, että algoritmeja ei tarvitse kirjoittaa erikseen *jokaiselle* säiliölle vaan yksi geneerinen funktiomalli toimii kaikkien säiliöiden kanssa.

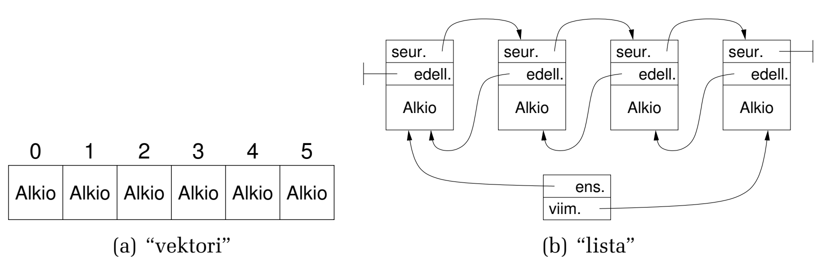
Tämä algoritmien geneerisyys tarkoittaa myös, että STL:n algoritmien käyttö ei rajoitu vain STL:n omiin säiliöihin. Ohjelmoija voi

kirjoittaa omia tietorakenteitaan, ja jos ne toteuttavat STL:n algoritmien asettamat vaatimukset, näitä algoritmeja voi käyttää käsittelemään myös ohjelmoijan omia tietorakenteita. Jos ohjelmoija esimerkiksi toteuttaa hajautustaulun tai binaaripuun, STL:n `find`-algoritmi pystyy etsimään alkioita niistäkin. Näin STL:n geneerisyys mahdollistaa myös STL:n "laajentamisen" omilla säiliötyypeillä.

10.1.3 Tietorakenteiden jaotteluperusteet

Kun kirjoissa ja oppilaitoksissa opetetaan tietorakenteita, niissä keskitytään luonnollisesti erityisesti tietorakenteiden sisäiseen rakenteeseen (tästä johtuu koko nimi "tieto*rakenne*"). Niinpä tiettyä tietorakennetta ajatellessaan suurimmalle osalle ohjelmoijista tulee mieleen juuri tietorakenteen sisäinen rakenne. Kuva 10.1 sisältää esimerkkejä mielikuvista, joita tyypilliselle ohjelmoijalle saattaisi tulla sanoista "vektori" ja "lista".

Olio-ohjelmoinnin kannalta tämä tapa mieltää tietorakenteet niiden rakenteen avulla on järjetön. Koko olio-ohjelmoinnin kantava ideahan on keskittyä olioiden käytössä rajapintoihin ja piilottaa sisäinen toteutustapa käyttäjältä. Niinpä tietorakenteiden käytön kannalta oleellista ei saisi olla se, *miten tietorakenne toteutetaan* vaan *miten tietorakennetta on tarkoitus käyttää.* Se, onko vektori toteutettu yhtenäisenä muistialueena ja sisältääkö lista linkkiosoittimet seuraavaan ja edelliseen alkioon, on osa tietorakenteen sisäistä toteutusta, jonka ei pitäisi näkyä tietorakenteesta ulospäin.



(a) "vektori" (b) "lista"

KUVA 10.1: Tietorakenteiden herättämiä mielikuvia

Olio-ohjelmoinnin kannalta olisi siis oleellista luokitella tietorakenteet niiden rajapintojen perusteella. Kun tietorakenteiden tarjoamia rajapintoja tarkastelee, huomaa kuitenkin pian, että varsin monilla tietorakenteilla on *periaatteessa aivan samanlainen rajapinta ja samanlaiset operaatiot.* Esimerkiksi vektoriin voi lisätä alkioita haluttuun paikkaan (jos oletetaan, että vektori kasvattaa kokoaan tarvittaessa), alkioita voi poistaa halutusta paikasta ja lisäksi halutun alkion voi hakea sen järjestysnumeron perusteella. Täsmälleen samat operaatiot ovat kuitenkin mahdollisia myös listan tapauksessa, ja on helppo keksiä myös muita tietorakenteita, joissa nämä operaatiot ovat mahdollisia.

Samanlaisista operaatioista huolimatta on kuitenkin selvää, että vektori ja lista ovat jollain lailla oleellisesti erilaisia tietorakenteita. Tämä ero on siinä, että eri tietorakenteiden sinänsä samanlaisten operaatioiden *tehokkuus* saattaa olla erilainen. Joillekin tietorakenteille uusien alkioiden lisääminen on nopeaa, toisille hidasta. Nopeus saattaa myös riippua siitä, *mihin kohtaan* uusi alkio lisätään. Vastaavasti tietyn alkion hakemisen ja muiden operaatioiden tehokkuus riippuu tietorakenteesta.

Esimerkiksi alkion lisääminen listan tietyn alkion perään on nopea operaatio, koska listan täytyy vain luoda uusi alkio ja linkittää se osoittimia muuttamalla mukaan listaan. Vastaava lisäysoperaatio vektorille on kuitenkin hidas, koska uudelle alkiolle täytyy tehdä tilaa esimerkiksi siirtämällä kaikkia lisäyskohdan jälkeen tulevia alkioita yhdellä eteenpäin. Vastaavasti halutun järjestysnumeron omaavan alkion haku vektorista on nopeaa, koska vektorin alkiot sijaitsevat peräkkäisissä muistiosoitteissa. Listalle vastaava operaatio taas on hidas, koska ainoa tapa etsiä alkio listasta on lähteä sen ensimmäisestä alkiosta ja seurata seuraavaan alkioon osoittavaa linkkiä tarvittavan monta kertaa.

Edellä mainitut asiat on otettu STL:ssä huomioon. Sen säiliöt on jaettu kahteen pääkategoriaan, joihin kuuluvien säiliöiden rajapinnat ovat keskenään lähestulkoon samanlaiset. Sen sijaan rajapinnan operaatioilta vaadittu tehokkuus vaihtelee säiliöstä toiseen. STL pyrkii myös estämään säiliöiden tehottoman käytön. Jokin operaatio saattaa puuttua kokonaan tietyn tyyppisestä säiliöstä, jos sen toteuttaminen tehokkaasti tällaiselle säiliölle ei olisi mahdollista. Säiliöiden operaatioiden tehokkuusvaatimukset on dokumentoitu C++-standardissa varsin tarkasti, kun taas niiden sisäiseen toteutukseen ei oteta lainkaan

kantaa. STL:n säiliöiden nimet on kuitenkin valittu niin, että niiden tehokkuusvaatimukset ovat nimeen nähden "luonnolliset" (esimerkiksi `list`-tietorakenteeseen on nopeaa lisätä uusia alkioita).

Kun ohjelmassa tulee tarve käyttää jotain tietorakennetta, ohjelmoijan tulisi ensimmäisenä miettiä ohjelman tehokkuusvaatimuksia. Niiden operaatioiden, joita tietorakenteelle suoritetaan usein, tulisi olla nopeita. Sen sijaan harvoin suoritettavien operaatioiden tehokkuudella ei yleensä ole mitään väliä. Näiden tietojen perusteella ohjelmoija voi sitten valita STL:stä tietorakenteen, jonka tehokkuusominaisuudet ovat ohjelman tarpeita vastaavat.

10.1.4 Tehokkuuskategoriat

Erilaisten operaatioiden tehokkuuden määrittely ja luokittelu ei ole yksinkertainen asia. Tehokkuuden mittaaminen milli- tai nanosekunneissa ei ole järkevää, koska tällainen "absoluuttinen" tehokkuus tietysti riippuu operaation toteutuksen tehokkuuden lisäksi myös käytetyn tietokoneen nopeudesta. Todellinen nopeus riippuu myös esimerkiksi tietorakenteisiin talletettujen alkioiden lukumäärästä ja koosta — on hitaampaa siirtää paljon tai suuria alkioita. Kuitenkin STL:n tapaisessa geneerisessä kirjastossa on tarve vertailla operaatioiden tehokkuuksia keskenään.

Tietojenkäsittelytieteessä yleisesti käytetty tehokkuuden mittari on se, miten tietyn operaation *nopeus muuttuu, kun tietorakenteen koko kasvaa*. Tällä tavoin ajateltuna ei ole väliä sillä, kuinka suuria alkiot ovat, kuinka monta niitä täsmälleen on tai kuinka nopea itse tietokone on. Oleellista on se, miten operaatio hidastuu, kun tietorakenteen alkioiden määrä kasvaa.

Esimerkkinä voidaan jälleen pitää vektoria ja listaa. Alkion lisääminen vektorin alkuun kestää sitä kauemmin, mitä suurempi vektori on. Tarkasti ottaen lisäämiseen kuluva aika on suoraan verrannollinen vektorin kokoon. Sen sijaan listan alkuun lisääminen kestää täsmälleen yhtä kauan, oli lista kuinka suuri tahansa. Vastaavasti vektorin alkion hakeminen järjestysnumeron perusteella on vakioaikainen operaatio kun taas listalle saman operaation aika on suoraan verrannollinen haetun alkion järjestysnumeroon, jota taas rajoittaa listan koko.

Operaation tehokkuus saattaa tietysti riippua myös tietorakenteen alkioiden sisällöstä, keskinäisestä järjestyksestä ja muista seikoista.

Operaation suoritusajalle voidaan näiden perusteella arvioida sekä ylä- että alaraja. Ohjelman tehokkuuden varmistamisen kannalta on kuitenkin usein oleellista, kuinka operaatio käyttäytyy *pahimmassa mahdollisessa tilanteessa.*

Usein tehokkuudessa kiinnostaa vain suoritusajan yleinen käyttäytyminen. Tällöin puhutaan usein operaation tehokkuuden **kertaluokasta** (*order of growth*). Sille voi laskea useita erilaisia tehokkuusmittoja, joille tietojenkäsittelytieteessä on omat merkintätapansa. Näistä merkintätavoista yleisimmät ovat

- **Θ-notaatio**, joka kertoo suoritusajan kertaluokan
- **O-notaatio**, joka kertoo suoritusajan “asymptoottisen ylärajan” — kertaluokan, jota suoritusaika ei varmasti ylitä, kun alkioiden määrä kasvaa tietyn rajan yli
- **Ω-notaatio**, joka kertoo suoritusajan “asymptoottisen alarajan” — kertaluokan, jota suoritusaika ei varmasta alita, kun tietty alkiomäärä ylitetään.

Näiden tehokkuuden merkintätapojen tarkka kuvaus jää tämän kirjan aihepiirin ulkopuolelle, mutta niistä löytyy tietoa lähes kaikista algoritmeja käsittelevistä kirjoista ja muista lähteistä (esimerkiksi “Introduction to Algorithms” [Cormen *ja muut*, 1990]).

On huomattava, että kertaluokkanotaatio kertoo vain, kuinka operaation suoritusaika käyttäytyy alkioiden määrän kasvaessa. Se ei kuitenkaan anna mitään tietoa varsinaisesta suoritusajasta. Esimerkiksi kahdesta lineaarisesta operaatiosta toinen voi olla paljon toista hitaampi — lineaarisuus kertoo vain, että molemmissa suoritusaika kasvaa suoraan suhteessa alkioiden määrään.

STL:n kannalta tärkein tehokkuuden mittari on ylärajan mittari O. Se kertoo, kuinka tehokas operaatio *ainakin* on. Esimerkiksi vakioaikainen tehokkuus $O(1)$ kertoo, ettei nopeus riipu alkioiden lukumäärästä. Lineaarinen tehokkuus $O(n)$ taas kertoo, että suoritusaika on suoraan verrannollinen alkioiden lukumäärään tai pienempi. Kuvaan 10.2 seuraavalla sivulla on kerätty tärkeimmät STL:ssä käytetyt tehokkuusluokat selityksineen ja O-merkintöineen.

Toinen STL:ssä jonkin verran käytetty mittari on keskimääräinen tehokkuus. Sitä käytetään joskus O-tehokkuuden rinnalla, jos huonoimman mahdollisen tapauksen tehokkuus eroaa oleellisesti keskimääräisestä tehokkuudesta. Esimerkiksi järjestysoperaatio `sort` on

Nimitys	O-notaatio	Selitys
Käännösaikainen (*compile-time*)	$O(0)$[a]	Operaatio suoritetaan käännösaikana eli se ei vaikuta suoritusaikaan.
Vakioaikainen (*constant*)	$O(1)$	Suoritusaika ei riipu alkioiden määrästä.
Amortisoidusti vakioaikainen (*amortized constant*)	$O(1)$	Operaatio on "käytännössä" vakioaikainen, yksittäistapauksissa kenties hitaampi.
Logaritminen (*logarithmic*)	$O(\log n)$	Suoritusaika on verrannollinen alkioiden määrän logaritmiin.
Lineaarinen (*linear*)	$O(n)$	Suoritusaika on suoraan verrannollinen alkioiden määrään.
	$O(n \log n)$	Suoritus hidastuu enemmän kuin lineaarisesti, ei kuitenkaan vielä neliöllisesti.
Neliöllinen (*quadratic*)	$O(n^2)$	Suoritusaika on verrannollinen alkioiden määrän neliöön.

[a]Vaikka merkintä $O(0)$ onkin teoreettisesti oikea, sitä ei yleensä käytetä.

KUVA 10.2: Erilaisia tehokkuuskategorioita

keskimääräiseltä tehokkuudeltaan $n \log n$, vaikka se pahimmassa tapauksessa voi olla esimerkiksi $O(n^2)$.

10.2 STL:n säiliöt

Kuten jo aiemmin tässä luvussa on todettu, STL:n säiliöiden rajapinnat muistuttavat suuresti toisiaan. Rajapintojensa puolesta STL:n säiliöt muutamaa poikkeusta lukuun ottamatta jakautuvat kahteen kategoriaan:

- **Sarjat** (*sequence*) ovat säiliöitä, joiden alkioita pystyy hakemaan niiden järjestysnumeron perusteella. Samoin alkioita voi lisätä haluttuun paikkaan ja poistaa siitä. Esimerkiksi taulukkotyyppi `vector` on tällainen sarja.

- **Assosiatiiviset säiliöt** (*associative container*) puolestaan perustuvat siihen, että alkioita haetaan säiliöstä **avaimen** (*key*) perusteella. Esimerkiksi puhelinluettelo muistuttaa assosiatiivista säiliötä — siinä numeron pystyy etsimään nopeasti nimen perusteella.

Vaikka näiden kahden kategorian säiliöt eroavatkin rajapinnoiltaan, niillä on myös yhteisiä rajapintaoperaatioita. Kaikilta säiliöiltä voi esimerkiksi kysyä jäsenfunktiolla `empty`, ovatko ne tyhjiä. Lisäksi niiden tarkan koon voi selvittää jäsenfunktiolla `size`.

Kuten mallit yleensäkin, myös STL:n säiliöt asettavat joitakin vaatimuksia tyyppiparametreilleen eli alkioidensa tyypille. Jotta tietyntyyppisiä alkioita varten voisi luoda säiliön, alkioiden tyypin täytyy toteuttaa kaksi ehtoa. Tyypillä tulee olla

- *kopiorakentaja,* joka luo alkuperäisen olion kanssa *samanlaisen* olion

- *sijoitusoperaattori,* jonka tuloksena sijoituksen kohteena olevasta oliosta tulee *samanlainen* sijoitetun olion kanssa.

Lisäksi STL:n assosiatiiviset säiliöt vaativat, että alkioiden avaimia voi myös kopioida ja sijoittaa ja lisäksi kahta avainta täytyy pystyä vertailemaan, jotta avaimet voidaan panna järjestykseen (oletusarvoisesti vertailu tehdään operaattorilla $<$).

Näistä vaatimuksista seuraa se, että *viitteet eivät kelpaa* STL:n säiliöiden alkioiksi tai avaimiksi. Viitteiden tapauksessahan viitteeseen sijoittaminen ei muuta viitettä viittaamaan toiseen olioon, vaan sijoittaa *viitteen päässä* olevat oliot. Riippuu käytetystä kääntäjästä, osaako se antaa virheilmoituksen, jos ohjelmassa yritetään luoda viitesäiliöitä.

Kaikki STL:n säiliöt varaavat itse lisää muistia tarvittaessa, kun niihin lisätään uusia alkioita. Kun säiliö tuhotaan, se vapauttaa kaiken varaamansa muistin. Sen sijaan ei ole varmaa, vapauttaako säiliö varaamaansa muistia *heti*, kun alkio poistetaan säiliöstä. Monet säiliöt nimittäin saattavat pitää muistia “varastossa” ja ottaa sen uudelleen käyttöön, kun säiliöön myöhemmin lisätään uusia alkioita.

10.2.1 Sarjat (“peräkkäissäiliöt”)

Sarjat (*sequence*) ovat säiliöitä, joissa alkiot sijaitsevat “peräkkäin” ja joissa jokaisella alkiolla on järjestysnumero. Alkioita voi selata järjestyksessä, ja halutun alkion voi hakea sen järjestysnumeron perusteella. Uusia alkioita voi lisätä säiliössä haluttuun paikkaan, ja vanhoja voi poistaa. STL tarjoaa kolme erilaista sarjasäiliötä: vector, deque ja list.

Kaikissa sarjoissa annetaan sarjan alkioiden tyyppi mallin tyyppiparametrina, siis esimerkiksi vector<**float**>, deque<**int**> ja list<string>. Tämän lisäksi ylimääräisenä tyyppiparametrina voi antaa sarjan muistinhallintaan käytettävän varaimen, mutta tätä mahdollisuutta ei käsitellä tässä enempää.

Sarjojen rajapinta on suurelta osin yhtenäinen. Uuden alkion voi kaikissa sarjoissa lisätä jäsenfunktiolla insert ja vanhoja alkioita voi poistaa jäsenfunktiolla erase. Lisäksi koko sarjan voi tyhjentää jäsenfunktiolla clear. Lisäksi osasta sarjoja löytyy vielä “ylimääräisiä” jäsenfunktioita sellaisia toimintoja varten, jotka kyseisessä sarjassa ovat erityisen nopeita. Esimerkiksi vector-tyypistä löytyy jäsenfunktio push_back, joka lisää uuden alkion taulukon loppuun. Saman toiminnon saisi tehtyä myös insert-jäsenfunktiolla, mutta silloin sille pitäisi erikseen kertoa, että lisäys tehdään taulukon loppuun.

Varsinaisten sarjojen lisäksi useimmat STL:n algoritmit suostuvat käsittelemään myös C++:n perustaulukoita (esimerkiksi **int** t[10]) kuten sarjoja, vaikka niiden ulkoinen rajapinta ei olekaan varsinaisesti sarjojen rajapintavaatimusten mukainen. Samoin merkkijonotyyppiä string voi käsitellä kuten merkeistä muodostuvaa vektoria.

Koska suurimmat erot eri sarjasäiliöiden välillä ovat tehokkuudessa, kuvaan 10.3 seuraavalla sivulla on kerätty tyypillisimpiä sarjojen tarjoamia operaatioita ja niiden tehokkuuksia.

Vektori — vector

Vektori (*vector*) on STL:n vastine taulukoille. Vektoreita voi indeksoida taulukoiden tapaan, ja tämä alkioiden haku järjestysnumeron perusteella on *vakioaikainen*. Uusien alkioiden lisääminen vektoriin on *lineaarinen* operaatio, paitsi jos uusi alkio lisätään vektorin loppuun, jolloin lisäys on *amortisoidusti vakioaikainen.* Samoin alkioi-

Operaatio	vector	deque	list
1./viim. alkio (front/back)	vakio	vakio	vakio
Mielivaltainen indeksointi ([], at)	vakio	vakio	—
1. alkion lisäys (push_front)	—[a]	vakio	vakio
1. alkion poisto (pop_front)	—[b]	vakio	vakio
Viim. alkion lisäys (push_back)	amort. vakio	vakio	vakio
Viim. alkion poisto (pop_back)	vakio	vakio	vakio
Mieliv. lisäys/poisto (insert/erase)	lineaar.	lineaar.	vakio
Sarjan koko/tyhjyys (size/empty)	vakio	vakio	lin./vakio
Sarjan tyhjentäminen (clear)	lineaar.	lineaar.	lineaar.

[a]Operaation voi suorittaa jäsenfunktiolla insert.
[b]Operaation voi suorittaa jäsenfunktiolla erase.

KUVA 10.3: Sarjojen operaatioiden tehokkuuksia

den poisto on *lineaarinen* operaatio, paitsi viimeisen alkion poisto on *vakioaikainen*.

Vektori saadaan käyttöön komennolla **#include** <vector>. Sen rajapinnassa on sarjojen perusrajapinnan lisäksi edellä mainittu indeksointi. Indeksoinnin voi tehdä joko normaaleilla hakasulkeilla [], jolloin mahdollista yli-indeksointia ei välttämättä tarkasteta, tai jäsenfunktiolla at, joka heittää poikkeuksen, jos annettu indeksi on liian suuri. Vektorin loppuun lisäystä ja poistoa varten ovat jäsenfunktiot push_back ja pop_back. Tämän lisäksi vektorin ensimmäisen ja viimeisen alkion voi lukea jäsenfunktioiden front ja back avulla.

Normaalin C++:n taulukon tapaan vektorin alkiot sijaitsevat muistissa peräkkäin.[ϒ] Kun vektoriin lisätään uusi alkio, sen kokoa täytyy kasvattaa. Tämä puolestaan tarkoittaa, että vektorin täytyy varata uusi suurempi muistialue, kopioida vanhat alkiot sinne ja vapauttaa vanha muistialue. Koska tämä operaatio on hidas (tarkasti ottaen lineaarinen), vektori ei varaa uutta muistia joka lisäyksellä. Sen sijaan vektori varaa aina kerralla tarvittavaa suuremman muistialueen, jonka alkuun se kopioi olemassa olevat alkiot. Kun uusia alkioita lisätään

[ϒ]Tarkasti ottaen tällaista vaatimusta ei löydy C++-standardista. Tämä on kuulemma kuitenkin ollut standardin kirjoittajien tarkoitus, ja kaikissa kääntäjissä vektori on toteutettu tällä tavalla. Lisäksi vaatimus lisätään standardiin seuraavan päivityksen yhteydessä.

vektorin loppuun, niille on jo valmiiksi muistia, joten uutta muistinvarausta ei tarvita. Kun varalla oleva muisti on kulutettu loppuun, vektorin täytyy taas varata suurempi muistialue ja niin edelleen.

Tällä tavoin uuden alkion lisääminen vektorin loppuun saadaan “käytännössä” vakioaikaiseksi, vaikka aina silloin tällöin muistinvaraus ja vanhojen alkioiden kopiointi hidastavatkin operaatiota. Vektorista löytyy myös jäsenfunktio `reserve`, jolla vektoria voi käskeä varautumaan annettuun määrään alkioita. Tällöin vektori varaa kerralla riittävästi muistia, jolloin uudelleenvarausta ei tarvita niin kauan, kuin alkioiden määrä pysyy annetuissa rajoissa.

Vektori on sopiva tietorakenne tilanteisiin, joissa tietorakennetta indeksoidaan paljon mutta uusia alkioita lisätään korkeintaan vanhojen perään ja vanhoja alkioita poistetaan vain lopusta. Käytännössä tämä tarkoittaa, että vektori kelpaa korvaamaan C++:n perustaulukot lähes kaikkialla. Listaus 10.1 sisältää esimerkin vektoreiden käytöstä. Funktio `laskeFibonacci` täyttää parametrina annetun vektorin annetulla määrällä Fibonaccin lukuja. (Fibonaccin luvuista kaksi ensimmäistä on 1 ja seuraavat alkiot saadaan aina laskemalla yhteen kaksi

```
1  #include <vector>
2  #include <algorithm>
3  using std::vector;
4  using std::min;
5
6  void laskeFibonacci(unsigned long lkm, vector<unsigned long>& taulukko)
7  {
8    taulukko.clear(); // Tyhjennä varmuuden vuoksi
9    taulukko.reserve(lkm); // Varaudu näin moneen alkioon
10
11   for (unsigned long i = 0; i < min<unsigned long>(lkm, 2); ++i)
12   {
13     // Ensimmäiset kaksi alkiota ovat 1
14     taulukko.push_back(1);
15   }
16   for (unsigned long i = 2; i < lkm; ++i)
17   {
18     // Loput alkiot kahden edellisen summa
19     taulukko.push_back(taulukko[i-2] + taulukko[i-1]);
20   }
21 }
```

LISTAUS 10.1: Fibonaccin luvut `vector`illa

edellistä alkiota.)

Pakka / kaksipäinen jono — deque

Kaksipäinen jono (*double-ended queue*) on taulukko, jossa *molempiin päihin* tehtävät lisäykset ja poistot ovat vakioaikaisia. Muualle tehtävät lisäykset ja poistot ovat lineaarisia. Kaksipäisen jonon nimi `deque` ääntyy "dek" samoin kuin englannin sana "deck", joka tarkoittaa korttipakkaa. Koska `deque` muistuttaa toiminnallisuudeltaankin jonkin verran korttipakkaa (kortteja on helppo lisätä ja poistaa pakan molemmista päistä), tässä teoksessa käytetään kaksipäisestä jonosta tästedes lyhempää nimitystä **pakka**.

Pakan rajapinta tarjoaa sarjojen perusrajapinnan lisäksi vielä vektorin rajapinnan eli operaatiot `[]`, `at`, `front`, `back`, `push_back` ja `pop_back`. Näiden operaatioiden tehokkuuskin on kertaluokaltaan sama kuin vektorissa. Koska myös pakan alkuun lisääminen ja sieltä poistaminen on nopeaa, pakassa on tätä varten vielä operaatiot `push_front` ja `pop_front`. Pakka saadaan käyttöön komennolla `#include <deque>`.

Koska pakan rajapinnassa on kaikki samat operaatiot kuin vektorissakin, ja operaatioiden tehokkuuskin on sama, pakkaa voi käyttää kaikkialla missä vektoriakin. Pakan indeksointi on kuitenkin käytännössä aika lailla vektorin indeksointia hitaampaa (vaikka molemmat ovatkin vakioaikaisia), joten suoritustehoa vaativissa tilanteissa vektori on parempi vaihtoehto.

Pakka eroaa vektorista edukseen siinä, että pakan alkuun lisäys ja siitä poisto ovat vakioaikaisia, kun ne vektorissa ovat lineaarisia. Pakka onkin kätevä tietorakenne toteuttamaan esimerkiksi puskureita, joissa tietoa lisätään toiseen päähän ja luetaan toisesta. Listaus 10.2 seuraavalla sivulla sisältää pakalla toteutetun luokan `LukuPuskuri`, johon voi lisätä rajoittamattoman määrän kokonaislukuja ja josta ne voi lukea pois samassa järjestyksessä. Lisäksi puskurista voi lukea annetussa kohdassa olevan alkion arvon.

Lista — list

Lista (*list*) on viimeinen STL:n sarjoista. Nimensä mukaisesti lista on tietorakenne, joka tehokkuudeltaan vastaa kahteen suuntaan linkitettyä listarakennetta. Alkioiden lisääminen ja poistaminen ovat *vakio-*

```
 1 #include <deque>
 2 using std::deque;
 3
 4 class LukuPuskuri
 5 {
 6 public:
 7   // (Rakentajat ja purkaja tyhjiä)
 8   void lisaa(int luku); // Lisää loppuun
 9   int lue(); // Lukee ja poistaa alusta
10   int katso(int paikka); // Lukee annetusta paikasta
11   bool onkoTyhja() const;
12
13 private:
14   deque<int> puskuri;
15 };
16
17 void LukuPuskuri::lisaa(int luku)
18 {
19   puskuri.push_back(luku);
20 }
21
22 int LukuPuskuri::lue()
23 {
24   int luku = puskuri.front();
25   puskuri.pop_front();
26   return luku;
27 }
28
29 bool LukuPuskuri::onkoTyhja() const
30 {
31   return puskuri.empty();
32 }
33
34 int LukuPuskuri::katso(int paikka)
35 {
36   return puskuri.at(paikka); // Ilman tarkistuksia: puskuri[paikka]
37 }
```

LISTAUS 10.2: Puskurin toteutus dequella

aikaisia riippumatta siitä, mihin kohtaan listalla lisäys tai poisto kohdistuu. Listasta voi myös lukea ensimmäisen tai viimeisen alkion vakioajassa mutta mielivaltaisen alkion lukeminen indeksoimalla *puuttuu kokonaan,* koska sitä ei saisi toteutetuksi tehokkaasti. Aliluvussa 10.3 esiteltävät iteraattorit antavat kuitenkin mahdollisuuden selata listaa läpi alkio kerrallaan.

Listan esittely luetaan komennolla `#include <list>`. Listan rajapinta tarjoaa sarjojen perusrajapinnan palvelut sekä operaatiot `back`, `push_back`, `pop_back`, `front`, `push_front` ja `pop_front`. Indeksointioperaatiot `[]` ja `at` puuttuvat.

Lista tarjoaa lisäksi erityisoperaatioita, jotka listan linkitetty rakenne tekee mahdolliseksi toteuttaa tehokkaasti. Jäsenfunktiolla `splice` voi vakioajassa siirtää yhden listan alkiot haluttuun paikkaan toisessa listassa. Samoin sillä voi vakioajassa siirtää osan listasta toiseen paikkaan samassa listassa. Lisäksi lista tarjoaa sitä varten optimoidut erikoisversiot eräistä STL:n algoritmeista.

Listan etuna vektoriin ja pakkaan verrattuna on se, että listojen yhdistäminen ja pilkkominen sekä alkioiden lisäys keskelle listaa ja poisto sieltä ovat vakioaikaisia. Vektorille ja pakalle kaikki nämä operaatiot ovat lineaarisia. Sen sijaan listaa ei voi indeksoida, joten sitä ei voi käyttää taulukkotyyppinä. Lista onkin parhaimmillaan puskureiden ja pinojen toteutuksessa sekä tilanteissa, joissa tietorakenteita tulee pystyä yhdistelemään.

10.2.2 Assosiatiiviset säiliöt

Assosiatiiviset säiliöt (*associative container*) eroavat sarjoista siinä, että alkioita ei lueta, lisätä tai poisteta niiden “sijainnin” tai järjestysnumeron perusteella vaan jokaiseen alkioon liittyy **avain** (*key*), jonka perusteella alkion voi myöhemmin hakea. Tästä tulee myös tämän säiliötyypin nimi: assosiatiiviset säiliöt määräävät assosiaation (“yhteyden”) avaimen ja alkion välille. Osassa assosiatiivisia säiliöitä alkio itse toimii myös avaimena, osassa avain ja alkio ovat erillisiä. STL tarjoaa neljä assosiatiivista säiliötä: `set`, `multiset`, `map` ja `multimap`.

Assosiatiivisille säiliöille annetaan tyyppiparametreina sekä avaimen että alkion tyyppi. Ylimääräisinä tyyppiparametreina on lisäksi mahdollista antaa avainten suuruusvertailuun käytettävä funktio sekä muistinhallintaan käytettävä varain. Jälleen nämä lisäominaisuudet jätetään tässä teoksessa läpikäymättä.

Rajapinnaltaan assosiatiiviset säiliöt muistuttavat toisiaan. Alkioita etsitään jäsenfunktiolla `find`, lisäätään jäsenfunktiolla `insert` ja poistetaan jäsenfunktiolla `erase`. Tämän lisäksi jotkut säiliötyypit tarjoavat lisäoperaatioita.

Kaikissa assosiatiivisissa säiliöissä alkioiden lisääminen, poistaminen ja hakeminen avaimen perusteella ovat tehokkuudeltaan logaritmisia operaatioita. Tämä onkin niiden etu sarjoihin verrattuna. Jos johonkin sarjasäiliöön talletettaisiin sekä avain että alkio, ainoa tapa tietyllä avaimella varustetun alkion etsimiseen olisi käydä läpi koko sarja alusta alkaen, ja tämä olisi tehokkuudeltaan lineaarinen operaatio.

Kuten kaikki muutkin säiliöt, assosiatiivisen säiliön alkiot voi selata läpi yksi kerrallaan aliluvussa 10.3 esiteltäviä iteraattoreita käyttäen. Tällöin alkiot käydään läpi avainten suuruusjärjestyksessä.

Joukko — `set`

Joukko (*set*) on yksinkertaisin assosiatiivisista säiliöistä. Joukossa alkio itse toimii avaimena, mistä johtuen joukolle annetaan tyyppiparametrina vain alkion tyyppi samoin kuin sarjoille. Esimerkiksi `set<char>` määrittelee joukon, johon talletetaan merkkejä. Joukot otetaan käyttöön komennolla `#include <set>`.

Käytännössä joukko vastaa aika hyvin "matemaattisen joukon" käsitettä. Siihen voi lisätä alkioita, niitä voi poistaa, ja joukolta voidaan kysyä, onko annettu alkio jo joukossa. Kaikki nämä operaatiot tehdään logaritmisessa ajassa. Samanarvoisia alkioita voi joukossa olla vain yksi kerrallaan kuten matematiikan joukossakin.

Joukko on käytännöllinen tietorakenne silloin, kun ohjelmassa täytyy ylläpitää jonkinlaista rekisteriä, johon lisätään alkioita sekalaisessa järjestyksessä, ja kun pitää pystyä nopeasti testaamaan, onko annettu alkio rekisterissä. Listaus 10.3 seuraavalla sivulla määrittelee esimerkkinä funktion `rekisterointi`, jolla voidaan rekisteröidä nimiä (merkkijonoja). Funktio ylläpitää nimijoukkoa `rekisteri` ja palauttaa **`true`**, jos rekisteröitävä nimi oli uusi eli sitä ei vielä ollut rekisterissä.

Assosiatiivisten säiliöiden operaatioilla on eräitä erityispiirteitä, jotka helpottavat niiden tehokasta käyttöä. STL:n optimointimahdollisuuksien yksityiskohtainen läpikäynti ei kylläkään ole mahdollista tässä teoksessa, mutta seuraava esimerkki havainnollistaa, millaisia mahdollisuuksia STL tarjoaa. Listauksen 10.3 rekisteröintifunktio

```
 1 #include <set>
 2 #include <string>
 3 using std::set;
 4 using std::string;
 5
 6 // Palauttaa true, jos rekisteröidään uusi nimi, muuten false
 7 bool rekisterointi(string const& nimi)
 8 {
 9   static set<string> rekisteri; // Staattinen, säilyy kutsujen välillä
10
11   if (rekisteri.find(nimi) == rekisteri.end())
12   {
13     // Ei ole ollut aiemmin
14     rekisteri.insert(nimi); // Lisää rekisteriin
15     return true;
16   }
17   else
18   {
19     // Nimi löytyi jo
20     return false;
21   }
22 }
```

LISTAUS 10.3: Nimien rekisteröinti `set`illä

toimii kyllä logaritmisessa ajassa mutta on siinä mielessä tehoton, että ensin rivillä 11 tutkitaan, löytyykö annettua merkkijonoa, ja sitten rivillä 14 sama merkkijono lisätään joukkoon, jos se ei jo ollut siellä. Näin alkion paikkaa joudutaan etsimään kaksi kertaa peräjälkeen.

Tilanne, jossa alkiota ensin etsitään ja sitten mahdollisesti lisätään se, on erittäin yleinen. Niinpä joukon `insert`-operaatio itse asiassa lisää annetun alkion *vain,* jos alkio ei jo ollut säiliössä. Lisäksi operaatio palauttaa paluuarvonaan `std::pair`-tyyppisen **`struct`**-tietorakenteen, jossa on myös tieto siitä, suoritettiinko lisäystä vai ei. Näin sama rekisteröintifunktio voidaan toteuttaa tehokkaammin yhdellä ainoalla `insert`-kutsulla. Listaus 10.4 seuraavalla sivulla näyttää tällaisen tehokkaamman toteutuksen.

Monijoukko — `multiset`

Monijoukko (**multiset**) eroaa tavallisesta joukosta siinä, että samanarvoisia alkioita voi olla monijoukossa useita. Monijoukolta voi alkion olemassaolon lisäksi kysyä, *montako* annetun arvoista alkiota moni-

```
 1 #include <set>
 2 #include <string>
 3 using std::set;
 4 using std::string;
 5
 6 // Palauttaa true, jos rekisteröidään uusi nimi, muuten false
 7 bool rekisterointi_optimoitu(string const& nimi)
 8 {
 9   static set<string> rekisteri; // Staattinen, säilyy kutsujen välillä
10
11   // insert palauttaa parin, jonka jälkimmäinen osa second ilmoittaa,
12   // tehtiinkö lisäys vai löytyikö alkio jo joukosta
13   return rekisteri.insert(nimi).second;
14 }
```

LISTAUS 10.4: Tehokkaampi versio listauksesta 10.3

joukossa on. Tämä tehdään jäsenfunktiolla `count` (itse asiassa `count` on myös tavallisen joukon rajapinnassa, jossa se palauttaa aina arvon 1). Monijoukon käyttöönotto tehdään komennolla `#include <set>` samoin kuin joukonkin.

Monijoukon käyttötarkoitus on lähes sama kuin joukon, mutta monijoukko mahdollistaa samanarvoisten alkioiden lisäämisen ja laskemisen. Listaus 10.5 sisältää uuden rekisteröintifunktion, joka lisää nimen rekisteriin ja ilmoittaa paluuarvonaan, kuinka monta kertaa nimi on lisätty rekisteriin.

```
 1 #include <set>
 2 #include <string>
 3 using std::multiset;
 4 using std::string;
 5
 6 // Palauttaa nimelle suoritettujen rekisteröintien lukumäärän
 7 unsigned long int monirekisterointi(string const& nimi)
 8 {
 9   static multiset<string> rekisteri; // Staattinen, säilyy kutsujen välillä
10   rekisteri.insert(nimi);
11   return rekisteri.count(nimi);
12 }
```

LISTAUS 10.5: Nimirekisteröinti `multiset`illä

Assosiaatiotaulu — map

Assosiaatiotaulu (*map*) on tietorakenne, jossa avain ja alkio ovat erillisiä ja jossa avaimen perusteella voidaan hakea haluttu alkio. Assosiaatiotaulua voi myös ajatella "taulukkona", jossa indeksinä käytetään halutun tyyppistä avainta kokonaisluvun sijaan. Yhtä avainta kohden voi assosiaatiotaulussa olla vain yksi alkio.

Assosiaatiotaulu ottaa kaksi tyyppiparametria, jotka määräävät avaimen ja alkion tyypit. Esimerkiksi `map<string, double>` on taulu, josta voi etsiä liukulukuja avaimina toimivien merkkijonojen avulla. Vastaavasti `map<int,string>` antaa mahdollisuuden liittää merkkijonoihin kokonaisluvun, jonka perusteella merkkijonon voi myöhemmin hakea. Assosiaatiotaulut saa käyttöön komennolla `#include <map>`.

Assosiaatiotaulun rajapinnassa on kaikki assosiatiivisten säiliöiden operaatiot. Kuten muissakin assosiatiivisissa säiliöissä, alkioiden etsiminen, lisääminen ja poistaminen kestävät logaritmisen ajan. Koska assosiaatiotaulu muistuttaa taulukkoa, siihen on lisätty myös indeksointi `[]`. Tämä etsii alkion avaimen perusteella kuten `find`, mutta jos alkiota ei löydy, indeksointi *lisää* automaattisesti tauluun uuden alkion, joka luodaan alkiotyypin oletusrakentajalla. Indeksointi palauttaa sitten joko löytyneen alkion tai tämän uuden alkion. Näin assosiaatiotaulua voi käyttää taulukkona, joka täyttyy tyhjillä alkioilla samalla, kun sitä indeksoidaan.

Listauksessa 10.6 on toteutettu nimien rekisteröinti assosiaatiotaululla. Nyt `rekisteri` onkin assosiaatiotaulu, joka liittää avaimena toimivaan merkkijonoon kokonaisluvun, joka kertoo montako kertaa

```
1  #include <map>
2  #include <string>
3  using std::map;
4  using std::string;
5
6  unsigned long monirekisterointi2(string const& nimi)
7  {
8    static map<string, unsigned long> rekisteri; // Säilyy kutsujen välillä
9    return ++rekisteri[nimi]; // Jos uusi nimi, lisätään autom. arvolla 0
10 }
```

LISTAUS 10.6: Nimirekisteröinti mapillä

merkkijono on rekisteröity. Funktio käyttää hyväkseen assosiaatiotaulujen indeksointia. Kun funktiolle annetaan uusi merkkijono, indeksointi ei löydä sitä taulusta. Tällöin tauluun lisätään uusi alkio, joka alustetaan oletusarvoonsa eli nollaksi. Jos taas merkkijono on jo rekisteröity, indeksointi palauttaa tiedon siitä, montako kertaa rekisteröinti on jo tehty. Rekisteröintilaskurin kasvattaminen saadaan nyt tehdyksi helposti kasvattamalla indeksoinnilla löydettyä arvoa yhdellä.

Assosiaatiomonitaulu — `multimap`

Assosiaatiomonitaulu (*multimap*) eroaa assosiaatiotaulusta samalla tavoin kuin monijoukko joukosta. Assosiaatiomonitaulussa yhtä avainta kohden voi olla useita alkioita. Tilanne vastaa esimerkiksi puhelinluetteloa, jossa yhtä nimeä kohti voi olla useita puhelinnumeroita.

Assosiaatiomonitaulun esittely luetaan komennolla **`#include`** `<map>` kuten tavallisenkin assosiaatiotaulun. Rajapinnan osalta assosiaatiomonitaulu muistuttaa assosiaatiotaulua, paitsi että indeksointia ei ole toteutettu. Syynä tähän on, että enää yhtä avainta kohden ei välttämättä löydy vain yhtä arvoa, jonka voisi palauttaa. Jäsenfunktio `find` palauttaa *jonkin* annettua avainta vastaavan alkion. Sopivista alkioista ensimmäisen voi hakea jäsenfunktiolla `lower_bound` ja viimeistä sopivaa seuraavan jäsenfunktiolla `upper_bound`, ja molemmat saa tietoonsa yhdellä kutsulla `equal_range`. Näitä ja aliluvussa 10.3 käsiteltäviä iteraattoreita käyttäen voi monitaulusta käydä läpi kaikki tiettyä avainta vastaavat alkiot.

Listaus 10.7 seuraavalla sivulla esittelee puhelinluetteloluokan, joka on toteutettu assosiaatiomonitaulua käyttäen. Jäsenmuuttuja `luettelo_` on monitaulu, jossa merkkijonon (nimen) perusteella voi hakea kokonaisluvun (puhelinnumeron). Koska monitaulun tyyppimäärittely on varsin pitkä, listauksessa rivillä 18 on määritelty tyyppinimi `LuetteloMap`. Puhelinluettelon jäsenfunktioiden toteutus löytyy listauksesta 10.8 sivulla 324. Jäsenfunktio `tulosta` käyttää iteraattoreita sopivien puhelinnumeroiden tulostamiseen, joten kyseistä koodia kannattaa tutkia tarkemmin vasta aliluvun 10.3 lukemisen jälkeen.

```
1  #include <map>
2  #include <string>
3  #include <iostream>
4  using std::multimap;
5  using std::pair;
6  using std::string;
7  using std::cout;
8  using std::endl;
9
10 class PuhLuettelo
11 {
12 public:
13    // Tyhjät rakentajat ja purkaja kelpaavat
14    void lisaa(string const& nimi, unsigned long numero);
15    unsigned long poista(string const& nimi);
16    void tulosta(string const& nimi) const;
17 private:
18    typedef multimap<string, unsigned long> LuetteloMap;
19    LuetteloMap luettelo_;
20 };
```

LISTAUS 10.7: `multimap`illä toteutettu puhelinluetteloluokka

10.2.3 Muita säiliöitä

Sarjojen ja assosiatiivisten säiliöiden lisäksi STL sisältää muutamia muita säiliöitä, jotka eivät tarkasti ottaen kuulu rajapinnaltaan kumpaankaan kategoriaan. Tällaisia säiliöitä ovat totuusarvovektori `vector<bool>`, bittivektori `bitset` sekä säiliösovittimet `queue`, `priority_queue` ja `stack`. Tämä aliluku esittelee lyhyesti näiden säiliöiden perusominaisuudet.

Totuusarvovektori — `vector<bool>`

Vektorimalli `vector` on varsin kätevä perustaulukkotyyppinä ja kelpaisi periaatteessa sellaisenaan totuusarvovektoriksi eli taulukoksi jonka alkiot ovat tyyppiä **bool**. Ongelmaksi muodostuu kuitenkin, että C++:ssa jokainen olio vie muistia vähintään yhden tavun verran, ja tämä rajoitus koskee myös perustyyppiä **bool** olevia alkioita. Näin tavallista vektoria käyttäen esimerkiksi 800 totuusarvon taulukko veisi muistia vähintään 800 tavua (todennäköisesti enemmän). Jokaisen to-

```
22 void PuhLuettelo::lisaa(string const& nimi, unsigned long numero)
23 {
24   luettelo_.insert(make_pair(nimi, numero)); // make_pair luo "parin"
25 }
26
27 unsigned long PuhLuettelo::poista(string const& nimi)
28 {
29   return luettelo_.erase(nimi); // Palauttaa poistettujen lukumäärän
30 }
31
32 void PuhLuettelo::tulosta(string const& nimi) const
33 {
34   // Etsi nimen perusteella ala- ja yläraja
35   pair<LuetteloMap::const_iterator, LuetteloMap::const_iterator>
36     alkuJaLoppu = luettelo_.equal_range(nimi);
37   // Käy läpi löytyneet alkio ja tulosta
38   cout << "Henkilö " << nimi << ":" << endl;
39   for (LuetteloMap::const_iterator i = alkuJaLoppu.first;
40        i != alkuJaLoppu.second; ++i)
41   {
42     cout << " " << i->second << endl;
43   }
44 }
```

LISTAUS 10.8: Puhelinluettelon toteutus

tuusarvon esittämiseen riittäisi kuitenkin jo yksi bitti, joten teoriassa 800 totuusarvon taulukon voisi saada mahtumaan 100 tavuun.

Tämä on esimerkki tilanteesta, jossa luokkamallien erikoistuksesta on hyötyä. STL:ssä on määritelty mallille `vector` erikoistus `vector<bool>`. Tämän erikoistuksen toteutus pakkaa totuusarvot muistin yksittäisiin bitteihin niin, että taulukko vie vähemmän muistia. Koska totuusarvovektori on toteutettu luokkamallin erikoistuksena, ei käyttäjän periaatteessa tarvitse edes tietää, että `vector<bool>` toteutukseltaan eroaa jotenkin tavallisista vektoreista.

Todellisuus ei kuitenkaan ole aivan näin ruusuinen. Koska totuusarvovektoria ei ole toteutettu aidosti erillisinä alkioina, sen rajapinnassa on pieniä eroavaisuuksia normaalin vektorin rajapintaan. Peruskäytössä nämä eroavaisuudet tuskin tulevat koskaan esille, mutta geneerisessä ohjelmoinnissa niillä saattaa olla merkitystä. Totuusarvovektorin eroista tavallisiin säiliöihin on kirjoitettu artikkeli "When Is a Container Not a Container?" [Sutter, 1999].

Bittivektori — bitset

Bittivektori (*bitset*) on tarkoitettu binaaristen bittisarjojen käsittelyyn. Rajapinnaltaan se ei itse asiassa ole edes säiliö, mutta C++-standardissa se jostain syystä esitellään luvun "assosiatiiviset säiliöt" alla. Bittivektorin saa käyttöön komennolla **#include** <bitset>.

Malliparametrinaan bittivektori saa yhden kokonaisluvun, joka kertoo kuinka monta bittiä vektori sisältää. Esimerkiksi bitset<64> määrittelee 64 bitin vektorin. Vektorin koko pysyy vakiona koko ajan, ja sen bitit alustetaan nollaksi.

Bittivektorin rajapinnassa on tyypillisiä binaarisia operaatioita. Vektorin yksittäisiä bittejä voi asettaa ykkösiksi ja nolliksi tai kääntää päinvastaisiksi. Lisäksi kahdelle samankokoiselle vektorille voi tehdä binaariset operaatiot &= (and), |= (or), ^= (xor) ja flip tai ~ (not). Vektorin bittejä voi myös siirtää halutun määrän oikealle tai vasemmalle operaattoreilla >>= ja <<=. Kaiken kukkuraksi bittivektori tarjoaa mahdollisuuden ykkösbittien laskemiseen, bittivektorin muuttamiseen kokonaisluvuksi ja takaisin sekä joitain muita erityisoperaatioita.

Säiliösovittimet

Kuten jo aiemmin on mainittu, STL tarjoaa joukon **säiliösovittimia** (*container adaptor*), jotka eivät itsessään ole säiliöitä, mutta joiden avulla säiliön rajapinnan saa "sovitetuksi toiseen muottiin". Säiliösovittimia on STL:ssä seuraavat kolme:

- **Pino** stack on luokkamalli, jonka rajapinnassa on pinon käsittelyyn tarvittavat operaatiot empty, size, top, push ja pop. Näistä push lisää uuden alkion pinon päälle, top palauttaa päällimmäisen alkion ja pop poistaa sen. Pinon esittely luetaan komennolla **#include** <stack>.

  Tyyppiparametrina pinolle annetaan alkioiden tyyppi ja säiliö, jota käyttäen pino toteutetaan. Esimerkiksi stack<**int**, list<**int**> > määrittelee pinon, joka on sisäisesti toteutettu listana.[♉] Pinon oletustoteutuksena on deque, joten pelkkä stack<**int**> tuottaa pakalla toteutetun pinon.

[♉] Huomaa syntaksissa sanaväli kahden loppukulmasulkeen välillä! Ilman sitä kääntäjä luulee, että on kyse operaattorista >>, ja antaa usein lähes käsittämättömän virheilmoituksen.

- **Jono** queue on samoin luokkamalli, joka tarjoaa rajapinnassaan jonolle tyypilliset operaatiot empty, size, front, back, push ja pop. Jonossa push lisää uuden alkion jonon perään ja pop poistaa alkion jonon alusta. Operaatio front lukee alkion jonon alusta ja back lopusta. Jonon saa käyttöönsä komennolla **#include** <queue>.

  Jonon luomisen syntaksi on aivan sama kuin pinonkin tapauksessa, eli queue<string, list<string> > luo listana toteutetun merkkijonojonon (☺) ja queue<**int**> kokonaislukujonon, joka on toteutettu pakan avulla.

- **Prioriteettijono** priority_queue on muuten kuin jono, mutta alkiot sijoitetaan suuruusjärjestykseen. Näin prioriteettijonosta luetaan front operaatiolla jonon pienin alkio, pop poistaa pienimmän alkion ja niin edelleeen. Prioriteettijonon syntaksi on sama kuin muidenkin säiliösovittimien, mutta sen toteutuksena ei voi käyttää listaa vaan toteutuksen on tuettava mielivaltaista indeksointia. Oletustoteutuksena prioriteettijonolla on vektori, ja sen esittely luetaan samalla komennolla **#include** <queue> kuin jononkin.

10.3 Iteraattorit

Ohjelmoinnissa on hyvin tavallista käydä tietorakenteen alkioita läpi järjestyksessä yksi kerrallaan. Vektoreiden ja pakkojen tapauksessa tämä onnistuu helposti indeksoinnin avulla, mutta esimerkiksi listoilla ja assosiatiivisilla säiliöillä ei ole nopeaa tapaa hakea annetun järjestysnumeron määräämää alkiota. Lisäksi indeksointi on vektoreiden ja pakkojenkin tapauksessa tarpeettoman tehoton operaatio, koska indeksoinnissa alkioiden laskeminen “aloitetaan aina alusta” eli indeksi ilmoittaa halutun alkion sijainnin suhteessa tietorakenteen alkuun.

Tyypillinen ratkaisuyritys ongelmaan on lisätä läpikäymiseen tarvittavat operaatiot itse tietorakenteeseen. Esimerkiksi listaluokasta voisi löytyä operaatiot annaEnsimmainen, annaSeuraava ja onkoLoppu, joiden avulla listan alkiot saisi käydyksi läpi. Tällöin lista muistaisi itse, missä alkiossa läpikäyminen on sillä hetkellä menossa. Tässä ratkaisuyrityksessä on kuitenkin kaksi ongelmaa:

- Koska listan täytyy sisältää tieto senhetkisestä paikasta, listaolion tila koostuu *sekä* listan alkioista *että* läpikäyntipaikasta. Tällöin operaatiot `annaEnsimmainen` ja `annaSeuraava` muuttavat väistämättä listan tilaa eivätkä näin ollen voi olla vakiojäsenfunktioita (aliluku 4.3). Tämä puolestaan tarkoittaa sitä, että vakioviitteen päässä olevaa listaa ei voi selata läpi, koska vakioviitteen läpi saa kutsua vain sellaisia jäsenfunktioita, jotka eivät muuta oliota. Vakioviitteiden käyttö parametreina on niin yleistä, että tämä rajoitus aiheuttaa suuria ongelmia.

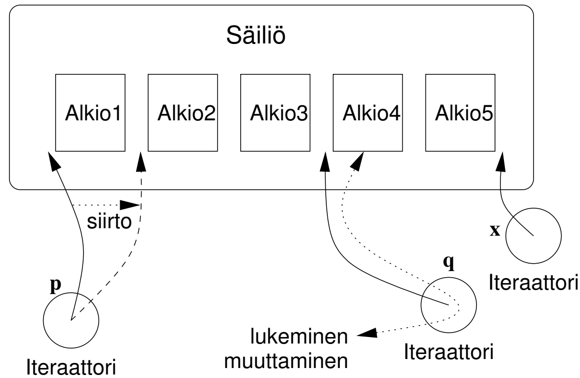
- Toinen ongelma on, että varsin usein listaa pitäisi pystyä käymään läpi *kahdesta kohtaa yhtä aikaa.* Esimerkiksi voi olla tarpeen lukea listaa samanaikaisesti alusta loppuun ja lopusta alkuun ja vertailla alkioita keskenään. Koska listaolio muistaa vain yhden paikan listalla, tällaista "limittäistä" läpikäyntiä ei voi tehdä.

Ratkaisu tähän läpikäyntiongelmaan löytyy käyttämällä olio-ohjelmoinnin perusperiaatteita. Koska listan alkioiden säilyttäminen ja läpikäyntipaikan muistaminen ovat selvästi erillisiä asioita, voidaan luoda *kaksi* luokkaa. Toinen on varsinainen lista, joka ei sisällä mitään paikkatietoa. Toinen on "kirjanmerkki", joka vain muistaa, missä kohtaa listaa ollaan läpikäymässä. Tällainen rakenne on aliluvussa 9.3.2 esitelty suunnittelumalli Iteraattori.

10.3.1 Iteraattoreiden käyttökohteet

STL:ssä iteraattorin käsite on erittäin tärkeä. Jokaista säiliötyyppiä kohti STL tarjoaa myös iteraattorityypin, jonka avulla säiliön alkiot voi käydä läpi. Iteraattoria voi ajatella kirjanmerkkinä, joka muistaa tietyn paikan tietyssä säiliössä. Iteraattoria voi siirtää säiliön sisässä, ja sen "läpi" voi myös lukea ja muuttaa säiliön alkioita. Kuva 10.4 seuraavalla sivulla havainnollistaa iteraattoreiden toimintaa.

Iteraattoreita käytetään kaikkialla STL:ssä ilmoittamaan tiettyä paikkaa säiliössä. Esimerkiksi poisto-operaatio `erase` ottaa parametrikseen iteraattorin, joka ilmoittaa missä kohdassa oleva alkio poistetaan. Tällöin poistettava alkio on iteraattorin osoittaman välin *oikealla puolella.* (Tarkasti ottaen "oikea" ja "vasen" ovat tietysti tietokoneessa järjettömiä termejä. Sanonnalla "oikealla puolella" tarkoite-



KUVA 10.4: Iteraattorit ja säiliöt

taan tässä *jälkimmäistä* niistä alkioista, joiden väliin iteraattori osoittaa.)

Iteraattoreita käytetään paikan ilmaisemiseen myös, kun uusia alkioita lisätään säiliöön. Jos säiliössä on jo aiemmin n alkiota, uuden alkion voi lisätä $n+1$ eri paikkaan. Näin ollen iteraattorilla pitää myös olla $n+1$ paikkaa, johon se voi osoittaa. Tämä saadaan aikaan, kun iteraattori voi osoittaa minkä tahansa kahden alkion väliin ja lisäksi säiliön kumpaankin päähän eli ensimmäisen alkion vasemmalle puolelle ja viimeisen alkion oikealle puolelle. Kuvan 10.4 oikeanpuoleisin iteraattori `x` havainnollistaa tätä. Uusi alkio lisätään aina iteraattorin osoittaman välin *oikealle* puolelle. Näin säiliön loppuun osoittavan iteraattorin avulla lisätty alkio lisätään todella säiliön loppuun.

Kun iteraattorin läpi luetaan alkion arvo tai muutetaan sitä, operaatio kohdistuu aina iteraattorin *oikealla* puolella olevaan alkioon samoin kuin alkion poistamisessakin. Säiliön loppuun osoittavan iteraattorin oikealla puolella ei kuitenkaan ole alkiota. Se onkin vain erityinen "loppumerkki", ja ohjelmoijan on pidettävä huoli siitä, ettei säiliön loppuun osoittavan iteraattorin läpi yritetä lukea tai kirjoittaa.

Säiliön loppuun osoittavaa iteraattoria käytetään myös merkkinä siitä, että operaatio ei onnistunut. Esimerkiksi assosiatiivisten säiliöiden hakuoperaatio `find` palauttaa paluuarvonaan iteraattorin, jonka oikealla puolella löytynyt alkio on. Mikäli halutunlaista alkiota ei löy-

tynyt, `find` palauttaa säiliön loppuun osoittavan iteraattorin merkkinä epäonnistumisesta.

Kahta iteraattoria voi myös käyttää määräämään tietyn **välin** (*range*) säiliössä. Tällä tarkoitetaan niitä alkioita, jotka jäävät iteraattoreiden väliin. Esimerkiksi kuvassa 10.4 iteraattorit `p` ja `q` määräävät välin, johon kuuluvat alkiot 1, 2 ja 3. Jos iteraattorit osoittavat samaan kohtaan, niiden välillä ei ole alkioita ja niiden määräämä väli on tyhjä. Välien avulla säiliöstä voidaan esimerkiksi poistaa useita alkioita kerrallaan. Lisäksi STL:n algoritmit ottavat yleensä parametreikseen nimenomaan iteraattorivälejä, jolloin algoritmin voi kohdistaa vain osaan säiliötä.

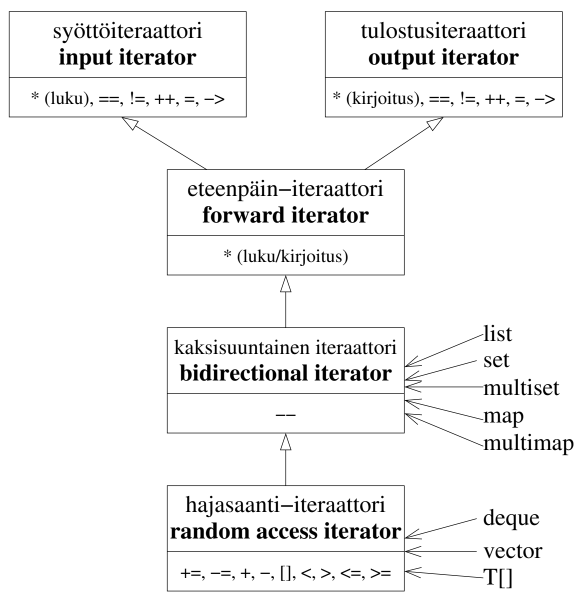
10.3.2 Iteraattorikategoriat

Erilaiset säiliöt tarjoavat erilaisia mahdollisuuksia siirtää iteraattoria nopeasti paikasta toiseen. Esimerkiksi yhtenäisellä muistialueella toteutetun vektorin tapauksessa iteraattorin saa vakioajassa siirretyksi kuinka paljon tahansa eteen- tai taaksepäin. Sen sijaan linkkiketjuilla toteutetussa listassa iteraattorin saa siirretyksi vakioajassa vain yhden askeleen verran.

STL:n suunnitteluperiaatteena on ollut, että *kaikkien* iteraattoreille tehtyjen operaatioiden täytyy onnistua vakioajassa. Tällä tavoin voidaan varmistua siitä, että STL:n algoritmit toimivat luvatulla tehokkuudella siitä riippumatta, millaisia iteraattoreita niille annetaan parametreiksi. Iteraattorit voidaan jakaa erilaisiin kategorioihin sen mukaan, millaisia vakioaikaisia operaatioita ne pystyvät tarjoamaan.

Kuva 10.5 seuraavalla sivulla näyttää STL:n iteraattorikategoriat. Kuvaan on merkitty myös, mihin kategoriaan kuuluvia iteraattoreita eri STL:n säiliöt tarjoavat. Vaikka kuva onkin piirretty UML-tyyliin periytymistä käyttäen, C++-standardi ei vaadi, että erilaiset iteraattorit on todellisuudessa periytetty toisistaan. Riittää, että niiden rajapinnat ovat kuvan mukaiset. STL:n iteraattorikategoriat ovat

- **syöttöiteraattori** (*input iterator*). Iteraattorin läpi voi vain *lukea* alkioita mutta ei muuttaa. Lisäksi iteraattoria voi siirtää yhden askelen kerrallaan *eteenpäin*. Iteraattorin `p` tarjoamat operaatiot ovat
  - `*p` : Iteraattorin osoittaman alkion arvon lukeminen



KUVA 10.5: Iteraattorikategoriat

- `p->` : Iteraattorin osoittaman alkion jäsenfunktion kutsuminen tai **`struct`**-kentän lukeminen (vertaa osoittimet)
- `++p` : Iteraattorin siirtäminen yhdellä eteenpäin
- `p++` : Kuten `++p`, mutta palauttaa paluuarvonaan iteraattorin, joka osoittaa *alkuperäiseen* paikkaan♊
- `p=q` : Iteraattorin sijoittaminen toiseen samanlaiseen
- `p==q`, `p!=q` : Sen vertailu, osoittavatko iteraattorit samaan paikkaan.

• **tulostusiteraattori** (*output iterator*). Iteraattori on muuten kuin lukuiteraattori, mutta sen läpi voi vain *muuttaa* alkioita, ei lukea. Muuttaminen tapahtuu syntaksilla `*p=x`.

♊Iteraattoreissa syntaksi `++p` on suositeltavampi kuin `p++` silloin, kun paluuarvolla ei ole merkitystä. Tähän on syynä tehokkuus: `p++` joutuu palauttamaan *kopion* iteraattorin alkuperäisestä arvosta, kun taas `++p` voi palauttaa vain viitteen iteraattoriin itseensä, jolloin kopiointia ei tarvita. Tällaisia tehokkuusasioita käsiteltiin aiemmin aliluvuissa 7.3 ja 7.1.

- **eteenpäin-iteraattori** (*forward iterator*). Iteraattorin läpi voi *lukea ja muuttaa* alkioita, ja lisäksi iteraattoria voi siirtää yhdellä eteenpäin. Eteenpäin-iteraattorin rajapinta on yhdistelmä luku- ja tulostusiteraattorin rajapinnoista.
- **kaksisuuntainen iteraattori** (*bidirectional iterator*). Iteraattori on muuten kuin eteenpäin-iteraattori, mutta se voi myös siirtyä yhden askelen kerrallaan *taaksepäin*. Uudet operaatiot ovat
  - `--p` : Iteraattorin siirtäminen yhdellä taaksepäin
  - `p--` : Kuten `--p`, mutta palauttaa paluuarvonaan iteraatorin, joka osoittaa *alkuperäiseen* paikkaan.
- **hajasaanti-iteraattori** (*random access iterator*). Iteraattori on kuin kaksisuuntainen iteraattori, mutta sitä voi siirtää kerralla mielivaltaisen määrän eteen- tai taaksepäin. Lisäksi iteraattoria indeksoimalla voi lukea ja muuttaa muitakin alkioita kuin iteraattorin oikealla puolella olevaa. Hajasaanti-iteraattoreita voi vertailla keskenään sen selvittämiseksi, kumpi on säiliössä ensimmäisenä. Lisäksi kahdesta hajasaanti-iteraattorista voi laskea, montako alkiota niiden välissä on. Näihin tarvittavat operaatiot ovat
  - `p+=n` : Iteraattorin siirtäminen `n` askelta eteenpäin (taaksepäin, jos `n` on negatiivinen)
  - `p-=n` : Iteraattorin siirtäminen `n` askelta taaksepäin (eteenpäin, jos `n` on negatiivinen)
  - `p+n` : Tuottaa uuden iteraattorin, joka osoittaa `p`:stä `n` askelta eteenpäin (taaksepäin, jos `n` on negatiivinen)
  - `p-n` : Tuottaa uuden iteraattorin, joka osoittaa `p`:stä `n` askelta taaksepäin (eteenpäin, jos `n` on negatiivinen)
  - `p[n]` : Sen alkion lukeminen tai muuttaminen, joka on `p`:n osoittamasta `n` askelta eteenpäin (taaksepäin, jos `n` on negatiivinen)
  - `p-q` : Kahden iteraattorin erotus ilmoittaa, kuinka monta askelta `p`:stä eteenpäin `q` osoittaa (jos `q` osoittaa paikkaan *ennen* `p`:tä, tulos ilmoitetaan negatiivisena)

– `p<q`, `p<=q`, `p>q`, `p>=q` : Iteraattori on toista iteraattoria “pienempi”, jos sen osoittama paikka on säiliössä ennen toisen osoittamaa paikkaa.

Iteraattoreiden operaatioiden syntaksi on tarkoituksella valittu sellaiseksi, että se vastaa C++:n ja C:n osoitinaritmetiikkaa, jota voi suorittaa taulukkoihin osoittavilla osoittimilla. Iteraattorit ovat itse asiassa vain osoitinaritmetiikan yleistys mielivaltaisille tietorakenteille. Vastaavasti C++:n perustaulukoihin osoittavat osoittimet kelpaavat iteraattoreiksi STL:n algoritmeissa.

10.3.3 Iteraattorit ja säiliöt

Iteraattorit liittyvät kiinteästi säiliöihin, joten STL:ssä itse iteraattoriluokat ja iteraattoreiden luominen on siirretty säiliöluokkien sisälle. Jokainen STL:n säiliöluokka määrittelee kaksi luokan sisäistä tyyppiä `iterator` ja `const_iterator`. Näistä `iterator` määrittelee kyseiselle säiliölle sopivan iteraattoriluokan. Tyyppi `const_iterator` puolestaan määrittelee vakio-iteraattorin, joka on muuten samanlainen kuin `iterator`, mutta sen läpi ei voi muuttaa säiliön sisältöä. Vakio-iteraattoreiden käyttö vastaa vakio-osoittimien ja vakioviitteiden käyttöä. Nämä iteraattoriluokat kuuluvat kuvan 10.5 näyttämiin iteraattorikategorioihin säiliötyypin mukaan.

Säiliöiden määrittelemiä iteraattorityyppejä käytetään kuin mitä tahansa aliluvussa 8.3 käsiteltyjä rajapintatyyppejä. Esimerkiksi kokonaislukuvektoriin osoittava iteraattori saadaan luoduksi syntaksilla

```
vector<int>::iterator p;
```

Tällaisista tyyppinimistä tulee usein pitkiä, joten niille kannattaa antaa paikallisesti lyhyempi nimi **`typedef`**-määrittelyllä:

```
typedef vector<int> Taulu;
typedef Taulu::iterator TauluIter;
TauluIter p;
```

Kun uusi iteraattori luodaan, se on oletusarvoisesti “tyhjä” eikä osoita minnekään. Tällöin sitä ei myöskään saa siirtää eikä sen läpi

saa yrittää lukea tai kirjoittaa. Jokaisen säiliön rajapinnassa ovat jäsenfunktiot `begin` ja `end`, joista `begin` palauttaa säiliön alkuun osoittavan “alkuiteraattorin”. Vastaavasti `end` palauttaa “loppuiteraattorin”, joka osoittaa säiliön loppuun. Nämä paluuarvot voi sitten sijoittaa talteen iteraattorimuuttujiin.

Erittäin tyypillinen iteraattoreiden käyttökohde on käydä säiliön alkioit läpi yksi kerrallaan. Tästä on esimerkki listauksessa 10.9. Riveillä 3–8 nollataan vektorin alkiot **`while`**-silmukassa. Tällöin luodaan iteraattori ja sijoitetaan siihen `begin`in palauttama iteraattori säiliön alkuun. Ensimmäisen alkion voi nyt lukea, tai sitä voi muuttaa `*`-operaattorin avulla. Seuraavaan alkioon pääsee `++`-operaatiolla. Joka kierroksella iteraattoria verrataan jäsenfunktion `end` palauttamaan loppuiteraattoriin, jotta tiedetään, milloin kaikki alkiot on käyty läpi. Tällöin on muistettava, että loppuiteraattori osoittaa paikkaan viimeisen alkion *jälkeen.*

Riveillä 13–17 puolestaan tulostetaan säiliön alkiot **`for`**-silmukassa. Koska alkioita ei ole tarkoitus muuttaa, kannattaa tässä käyttää vakio-iteraattoria. Listauksen esimerkissä on itse asiassa *pakko* käyttää vakio-iteraattoria, koska funktio saa parametrinaan *vakioviitteen* vek-

```
1  void nollaaAlkiot(vector<int>& vektori)
2  {
3    vector<int>::iterator i = vektori.begin(); // Alkuun
4    while (i != vektori.end()) // Toistetaan kunnes ollaan lopussa
5    {
6      *i = 0; // Nollataan alkio
7      ++i; // Siirrytään seuraavaan alkioon
8    }
9  }
10
11 void tulostaAlkiot(vector<int> const& vektori)
12 {
13   for (vector<int>::const_iterator i = vektori.begin();
14        i != vektori.end(); ++i)
15   {
16     cout << *i << " ";
17   }
18   cout << endl;
19 }
```

LISTAUS 10.9: Säiliön läpikäyminen iteraattoreilla

toriin. Tämän viitteen läpi vektoria ei saa muuttaa. Tämä on varmistettu STL:ssä niin, että vakioviitteen ja -osoittimen kautta kutsuttuna `begin` ja `end` palauttavat automaattisesti vakio-iteraattorin, jonka voi sijoittaa vain toiseen vakio-iteraattoriin.

Säiliöiden omat jäsenfunktiot käyttävät yksinomaan iteraattoreita paikan ilmaisemiseen (ainoana poikkeuksena on vektorin ja pakan indeksointi, jossa käytetään järjestysnumeroa). Vaikka iteraattoreiden käyttö vaatii totuttelua, niiden avulla säiliöiden käyttö käy varsin kätevästi. Esimerkiksi uuden alkion `3` lisääminen kokonaislukuvektorin `v` alkuun tehdään syntaksilla `v.insert(v.begin(), 3)`. Vastaavasti vektorin viidennen alkion poisto onnistuu iteraattoriaritmetiikkaa käyttäen kutsulla `v.erase(v.begin()+5)`.

10.3.4 Iteraattoreiden kelvollisuus

Periaatteessa iteraattorit osoittavat tiettyä *paikkaa* säiliössä eivätkä näin ollen ole sidoksissa itse säiliön alkioihin. Iteraattoreiden sisäinen toteutus riippuu kuitenkin käytännössä säiliöiden sisäisestä rakenteesta, ja iteraattorit sisältävät todennäköisesti osoittimia säiliöiden sisäiseen toteutukseen tai alkioihin.

Jos nyt säiliöstä poistetaan alkioita tai sinne lisätään uusia alkioita, voi säiliöstä riippuen sen sisäinen rakenne muuttua. Esimerkiksi vektori saattaa varata lisää tilaa ja siirtää alkiot sinne. Tämä voi puolestaan aiheuttaa sen, että iteraattorin sisäinen tieto ei enää olekaan ajan tasalla.

STL:ssä sanotaan, että iteraattori on **kelvollinen** (*valid*) niin kauan, kun se on käyttökelpoinen. Muutokset säiliössä saattavat aiheuttaa sen, että iteraattorista tulee **kelvoton** (*invalid*). Tällöin sanotaan, että jokin säiliön operaatio **mitätöi** (*invalidate*) tietyt iteraattorit. Tällaiselle kelvottomalle iteraattorille ainoat sallitut operaatiot ovat tuhoaminen ja uuden arvon sijoittaminen. Kaikki muut operaatiot aiheuttavat sen, että ohjelman käyttäytyminen on määrittelemätön. Iteraattoreiden lisäksi kelvollisuus koskee myös osoittimia ja viitteitä säiliön alkioihin. Jos muutos säiliössä siirtää esimerkiksi alkion toiseen paikkaan muistissa, myös kaikki osoittimet ja viitteet kyseiseen alkioon muuttuvat kelvottomiksi.

Iteraattoreiden muuttuminen kelvottomiksi riippuu säiliöstä ja siitä, millainen muutos siihen tehdään. Seuraavassa luettelossa on

lueteltu STL:n säiliöt ja säiliöön kohdistuvien muutosten vaikutus iteraattoreihin, viitteisiin ja osoittimiin.

- **Vektori:** Jos uuden alkion lisäämisessä ei tarvita muistin uudelleenvarausta (vektorille on varattu riittävästi tilaa `reserve`llä), mitätöityvät kaikki iteraattorit, osoittimet ja viitteet, jotka osoittavat lisäyspaikan jälkeisiin alkioihin. Jos uudelleenvaraus suoritetaan, mitätöityvät *kaikki* vektoriin osoittavat iteraattorit, osoittimet ja viitteet. Alkion poisto vektorista mitätöi kaikki iteraattorit, osoittimet ja viitteet poistopaikasta alkaen vektorin loppuun saakka.

- **Pakka:** Uuden alkion lisäys pakan alkuun tai loppuun mitätöi kaikki pakkaan osoittavat iteraattorit. Osoittimet ja viitteet sen sijaan säilyvät kelvollisina. Alkion poisto pakan alusta tai lopusta mitätöi vain heti poistettavan alkion vasemmalla puolella olevan iteraattorin sekä tietysti osoittimet ja viitteet poistettuun alkioon. Lisäys pakan keskelle tai poisto sieltä mitätöi kaikki pakkaan osoittavat iteraattorit, osoittimet ja viitteet.

- **Lista:** Alkion lisääminen ei mitätöi mitään. Alkion poisto mitätöi heti alkion vasemmalla puolella olevan iteraattorin sekä osoittimet ja viitteet poistettuun alkioon.

- **Assosiatiiviset säiliöt:** Lisäys ja poisto vaikuttavat samalla tavoin kuin listaan.

Iteraattoreiden kelvollisuuden huomioon ottaminen on äärimmäisen tärkeää ohjelmoinnissa. Koska kelvottomaan iteraattoriin kohdistetut operaatiot aiheuttavat ohjelman määrittelemättömän käyttäytymisen, kelvottomista iteraattoreista aiheutuvia virheitä on *erittäin* vaikea löytää. Moniin C++-ympäristöihin on saatavilla STL-toteutuksia, jotka osaavat "testitilassa" ollessaan antaa kelvottomien iteraattoreiden käytöstä välittömästi järkevän virheilmoituksen. Yksi tällainen STL-toteutus on STLport [Fomitchev, 2001].

Eri säiliöt mitätöivät iteraattoreitaan eri tavalla. Niinpä säiliöiden tehokkuuden lisäksi myös säiliön operaatioiden mitätöimisvaikutukset kannattaa ottaa huomioon ohjelmaan sopivaa säiliötyyppiä valittaessa.

10.3.5 Iteraattorisovittimet

Tavallisten iteraattoreiden lisäksi STL tarjoaa myös joukon **iteraattorisovittimia** (*iterator adaptor*). Nämä ovat "erikoisiteraattoreita", jotka käyttäytyvät jollain lailla tavallisista iteraattoreista poiketen. Iteraattorisovittimien perinpohjainen läpikäynti ei ole mahdollista tässä teoksessa, mutta tässä aliluvussa ne esitellään kuitenkin lyhyesti. Iteraattorisovittimet ovat hyviä esimerkkejä siitä, että iteraattoreiden käyttöalue on paljon laajempi kuin vain säiliöiden yksinkertainen läpikäynti. Iteraattorisovittimien avulla voidaan myös muunnella STL:n algoritmien toiminnallisuutta. Iteraattorisovittimet otetaan käyttöön komennolla `include <iterator>`.

- **Käänteisiteraattorit** (*reverse iterator*) ovat tavallisten iteraattoreiden "peilikuvia". Käänteisiteraattoreilla `++` siirtää iteraattoria taaksepäin ja vastaavasti `--` eteenpäin. Myös muut siirto-operaatiot on peilattu. Lisäksi luku ja kirjoitus käänteisiteraattorin kautta kohdistuvat iteraattorin osoittaman paikan *vasemmalle* puolelle. Jokaisesta STL:n säiliöstä saa käänteisiteraattorin jäsenfunktioilla `rbegin`, joka palauttaa käänteisiteraattorin säiliön loppuun (siis peilattuna alkuun), ja `rend`, joka antaa vastaavasti käänteisiteraattorin säiliön alkuun.

- **Lisäysiteraattorit/lisääjät** (*insert iterator*/*inserter*) ovat tulostusiteraattoreita, joiden läpi kirjoittaminen *lisää* säiliöön uuden alkion vanhan alkion muuttamisen sijasta. Niiden avulla saadaan STL:n algoritmit lisäämään alkioita säiliöihin. Uuden alkion lisääminen käy yksinkertaisesti syntaksilla `*p=x`. Tämän jälkeen lisäysiteraattoria täytyy vielä siirtää eteenpäin `++`-operaattorilla. Säiliön alkuun lisäävän lisäysiteraattorin saa funktiokutsulla `front_inserter(sailio)` ja loppuun lisäävän kutsulla `back_inserter(sailio)`. Annetun iteraattorin kohdalle alkioita lisäävän iteraattorin saa aikaan kutsulla `inserter(sailio, paikka)`.

- **Virtaiteraattorit** (*stream iterator*) ovat luku- tai tulostusiteraattoreita, jotka säiliöiden sijaan lukevat ja kirjoittavat C++:n tiedostovirtoihin. Näin esimerkiksi tiedostoja voi käyttää STL:n algoritmeissa säiliöiden tapaan. Esimerkiksi `cin`-virrasta kokonaislukuja lukevan lukuiteraattorin saa syntaksilla `istream_iterator<int>(cin)` ja merkkijonoja `cout`-vir-

taan pilkuilla erotettuina tulostava iteraattori luodaan kutsulla `ostream_iterator<string>(cout, ',')`.

STL:n valmiina tarjoamien iteraattorisovittimien lisäksi ohjelmoija voi kirjoittaa myös omia iteraattorityyppejään. Näin iteraattorit antavat varsin monipuolisen työkalun säiliöiden käsittelyyn.

10.4 STL:n algoritmit

STL tarjoaa monenlaisia algoritmeja säiliöiden käsittelyyn. Jo aiemmin on todettu, että nämä algoritmit on toteutettu irrallisina funktiomalleina jäsenfunktioiden sijaan. Tämä tarkoittaa sitä, että algoritmien täytyy saada kaikki tarvitsemansa tieto parametrien avulla. Yhdellekään STL:n algoritmille ei kuitenkaan anneta parametreina itse säiliöitä, vaan kaikki algoritmit ottavat parametreinaan iteraattoreita. Syitä tähän ehkä vähän yllättävään suunnitteluperiaatteeseen on useita:

- Iteraattoreiden avulla algoritmi saadaan toimimaan vain *osalle* säiliötä. Suurin osa STL:n algoritmeista ottaa parametreinaan kahden iteraattorin määräämän välin. Tämä voi olla joko koko säiliö (jos parametreina annetaan `begin`- ja `end`-kutsujen tuottamat iteraattorit) tai vain osa siitä.

- Iteraattorit mahdollistavat sen, että sama algoritmi tekee toimintonsa erilaisten säiliöiden kesken, koska itse säiliöiden tyyppiä ei tarvitse kertoa algoritmille. Esimerkiksi `merge`-algoritmin avulla voi yhdistää listan ja pakan sisällöt vektoriin.

- Iteraattorisovittimien avulla voi vaikuttaa algoritmin toimintaan. Esimerkiksi `find`-algoritmi etsii normaalisti ensimmäisen halutun arvoisen alkion. Kun sille annetaan normaalien iteraattoreiden sijaan käänteisiteraattoreita, se etsiikin viimeisen sopivan alkion. Samoin normaalisti `copy`-algoritmi korvaa säiliön alkiot toisen säiliön alkioilla. Jos käytetään lisäysiteraattoreita, `copy` kuitenkin lisää kopioitavat alkiot korvaamisen sijasta.

- Mikään ei estä ohjelmoijaa kirjoittamasta omia iteraattorityyppejään. Tällöin STL:n algoritmit toimivat myös niiden kanssa.

> Lähes mihin tahansa tietorakenneluokkaan on helppo kirjoittaa siihen sopivat iteraattoriluokat, joten STL:n algoritmit saa vähällä vaivalla sovitetuksi lähes mihin tahansa tietorakenteeseen. Iteraattoreiden avulla tämä onnistuu, vaikka itse tietorakenteen rajapinta ei olisikaan yhtenäinen STL:n säiliöiden rajapintojen kanssa.

Voidaan sanoa, että iteraattorit ovat ikään kuin liima algoritmien ja säiliöiden välissä. Koska algoritmit käyttävät iteraattoreita säiliöiden käsittelyyn, niiden tehokkuus riippuu myös iteraattoreiden tehokkuudesta. Tämä ei kuitenkaan ole ongelma, koska iteraattorit tarjoavat vain sellaisia operaatioita, jotka ovat vakioaikaisia. Eri iteraattorikategoriat tarjoavat näitä operaatioita eri määrän. Toimiakseen tehokkaasti algoritmit saattavat vaatia, että niille annettavien iteraattoreiden täytyy kuulua vähintään tiettyyn kategoriaan.

Esimerkiksi säiliön alkiot järjestävä `sort`-algoritmi vaatii, että sille annettujen iteraattoreiden täytyy olla hajasaanti-iteraattoreita. Mikäli näin ei ole, annetaan käännösaikainen virheilmoitus. Tästä rajoituksesta seuraa, että listaa ei voi järjestää `sort`-algoritmilla, koska se tarjoaa vain kaksisuuntaiset iteraattorit (tämän vuoksi lista tarjoaa järjestämisen jäsenfunktionaan). Sen sijaan `find`-algoritmille riittää, että annetut iteraattorit ovat vähintään lukuiteraattoreita.

Iteraattorikategorioiden avulla kääntäjä voi jo *käännösaikana* varmistua, että algoritmit pystyvät toteuttamaan tehokkuuslupauksensa. Mikäli algoritmille yritetään antaa iteraattori sellaiseen säiliöön, jonka operaatiot eivät ole algoritmille riittävän tehokkaita, algoritmi ei yksinkertaisesti käänny. Saatu virheilmoitus saattaa kylläkin kääntäjästä riippuen olla varsin kryptinen, kuten mallien yhteydessä ikävä kyllä usein tapahtuu.

Jotta STL:n algoritmeja pystyisi käyttämään, niiden esittelyt täytyy ensin ottaa käyttöön komennolla `#include <algorithm>`. Seuraavassa listassa on esitelty lyhyesti joitain STL:n algoritmeja. Listan tarkoituksena on antaa yleiskuva siitä, millaisia asioita algoritmeilla pystyy tekemään. Algoritmien tarkemman syntaksin ja kuvauksen voi lukea monista C++-kirjoista [Lippman ja Lajoie, 1997], [Stroustrup, 1997] tai erityisesti C++:n kirjastoja käsittelevistä kirjoista [Josuttis, 1999].

- `copy(alku, loppu, kohde)` kopioi välillä `alku`–`loppu` olevat alkiot iteraattorin `kohde` päähän. Se *korvaa* vanhat alkiot kopioi-

duilla, joten alkioiden *lisäämiseen* tarvitaan lisäysiteraattoreita. Operaatio on lineaarinen, `alku` ja `loppu` lukuiteraattoreita, `kohde` tulostusiteraattori.

- `find(alku, loppu, arvo)` etsii väliltä `alku`–`loppu` ensimmäisen alkion, jonka arvo on `arvo`, ja palauttaa iteraattorin siihen. Jos sopivaa alkiota ei löydy, palautetaan `loppu`. Tehokkuudeltaan operaatio on lineaarinen, `alku` ja `loppu` ovat lukuiteraattoreita. Algoritmille voi `arvon` sijaan antaa parametrina totuusarvon palauttavan funktion, joka kertoo onko sille annettu parametri halutunlainen.

- `sort(alku, loppu)` järjestää välillä `alku`–`loppu` olevat alkiot suuruusjärjestykseen. Algoritmin tehokkuus on *keskimäärin* $n \log n$, ja iteraattoreiden on oltava hajasaanti-iteraattoreita. `sort`-algoritmille voi myös antaa vertailufunktion ylimääräisenä parametrina.

- `merge(alku1, loppu1, alku2, loppu2, kohde)` edellyttää, että välien `alku1`–`loppu1` ja `alku2`–`loppu2` alkiot ovat suuruusjärjestyksessä. Algoritmi yhdistää kyseisten välien alkiot ja kopioi ne suuruusjärjestyksessä iteraattorin `kohde` päähän aivan kuten `copy`. Operaatio on lineaarinen, `alku`- ja `loppu`-iteraattorit ovat lukuiteraattoreita, `kohde` on tulostusiteraattori.

- `for_each(alku, loppu, funktio)` antaa jokaisen välillä `alku`–`loppu` olevan alkion vuorollaan `funktion` parametriksi ja kutsuu funktiota. Tehokkuudeltaan `for_each` on lineaarinen. Jos `alku` ja `loppu` ovat lukuiteraattoreita, `funktio` ei saa muuttaa parametrinaan saamansa alkion arvoa. Jos iteraattorit ovat eteenpäin-iteraattoreita, alkioita saa muuttaa.

- `partition(alku, loppu, ehtofunktio)` järjestää välillä `alku`–`loppu` olevat alkiot niin, että ensin tulevat ne alkiot, joilla `ehtofunktio` palauttaa **`true`**, ja sitten ne, joilla se palauttaa **`false`**. Tehokkuus algoritmissa on lineaarinen, iteraattoreiden tulee olla kaksisuuntaisia iteraattoreita.

- `random_shuffle(alku, loppu)` sekoittaa välillä `alku`–`loppu` olevat alkiot satunnaiseen järjestykseen. Sekoituksen tehokkuus

on lineaarinen, `alku` ja `loppu` hajasaanti-iteraattoreita. Algoritmille voi ylimääräisenä parametrina antaa oman satunnaislukugeneraattorin.

Listauksessa 10.10 on lyhyt esimerkki STL:n algoritmien käytöstä. Listauksen funktio ottaa ensin muistiin vektorin ensimmäisen ja viimeisen alkion arvot. Sen jälkeen se järjestää vektorin suuruusjärjestykseen. Se etsii vielä järjestetystä vektorista muistiin otettuja arvoja vastaavat alkiot. Lopuksi se poistaa näiden väliin jäävät alkiot eli alkiot, jotka ovat suurempia tai yhtä suuria kuin alkuperäinen ensimmäinen alkio mutta pienempiä kuin alkuperäinen viimeinen alkio. Operaatioista `find` ja `erase` ovat lineaarisia ja `sort` keskimäärin $n \log n$, joten koko funktion keskimääräiseksi tehokkuudeksi tulee $n \log n$.

10.5 Funktio-oliot

Monissa tapauksissa STL:n algoritmeille pitää iteraattoreiden lisäksi välittää myös tietoa siitä, miten algoritmin tulisi toimia. Esimerkiksi algoritmille `sort` voi antaa tarvittaessa tiedon siitä, miten alkioiden

```
1  #include <algorithm>
2  using std::find;
3  using std::sort;
4  #include <vector>
5  using std::vector;
6
7  void jarjestaJaPoista(vector<int>& vektori)
8  {
9    int eka = vektori.front(); // Ensimmäinen alkio talteen
10   int vika = vektori.back(); // Viimeinen alkio talteen
11   sort(vektori.begin(), vektori.end()); // Järjestä
12   vector<int>::iterator ekanpaikka =
13     find(vektori.begin(), vektori.end(), eka); // Etsi eka
14   vector<int>::iterator vikanpaikka =
15     find(vektori.begin(), vektori.end(), vika); // Etsi vika
16   // Poista alkiot eka ≤ alkio < vika
17   vektori.erase(ekanpaikka, vikanpaikka);
18 }
```

LISTAUS 10.10: Esimerkki STL:n algoritmien käytöstä

“suuruutta” vertaillaan keskenään. Vastaavasti algoritmi `find_if` etsii ensimmäisen alkion, joka toteuttaa algoritmille välitetyn ehdon.

STL:ssä on yleensä kaksi tapaa välittää algoritmien (ja assosiatiivisten säiliöiden) sisään tällaista “toiminnallisuutta”. Toinen tapa, **funktio-osoittimet** (*function pointers*), on jo C-kielestä peräisin. Niiden lisäksi STL:ssä käytetään usein **funktio-olioita** (*function objects*), joilla voidaan saada hieman yleiskäyttöisempiä ratkaisuja. Seuraavat aliluvut esittelevät näiden kahden tavan perusteet.

10.5.1 Toiminnallisuuden välittäminen algoritmille

Oletetaan, että ohjelmassa halutaan tulostaa kaikki kokonaislukuvektorin alkiot, jotka ovat arvoltaan alle 5. Tämä olisi tietysti mahdollista tehdä suhteellisen helposti **`for`**-silmukalla, mutta toisaalta STL:stä löytyy valmiina algoritmeja halutunlaisten alkioiden etsimiseen. Jos tarkoituksena olisi hakea alkiot, joiden arvo on *yhtä suuri* kuin 5, löytyisi ensimmäinen tällainen alkio yksinkertaisesti kutsulla

```
find(vektori.begin(), vektori.end(), 5)
```

Sen sijaan 5:ttä pienempien alkioiden etsiminen on hieman vaativampaa. STL:stä *ei* löydy valmista algoritmia `find_less`, koska erilaisia tällaisia hakualgoritmeja olisi niin monta erilaista, ettei niiden koodaaminen erikseen olisi järkevää. Sen sijaan STL:ssä on algoritmi `find_if`, jolle voidaan kertoa, millaista alkiota halutaan. Helpoin tapa käyttää tätä algoritmia on välittää sille kolmantena parametrina **funktio-osoitin** (*function pointer*), joka osoittaa funktioon jota käytetään alkioiden testaamiseen.

Funktio-osoitinta voi ajatella tavallisena osoittimena, joka muuttujaan tai olioon osoittamisen sijaan osoittaakin johonkin ohjelman funktioon. Tällaisen osoittimen saa luotua normaalilla syntaksilla `&funktionimi`, ja funktio-osoittimen läpi funktiota kutsutaan ikään kuin osoitin itse olisi kyseinen funktio. Funktio-osoittimen tyyppi määrää millaisia parametreja ottaviin funktioihin osoitin voi osoittaa. Samoin tyyppi määrää myös osoitettavan funktion paluutyypin. Tässä teoksessa ei mennä funktio-osoittimien käytön yksityiskohtiin, mutta mainittakoon, että funktio-osoittimien lisäksi C++ tarjoaa myös mahdollisuuden **jäsenfunktio-osoittimiin** (*member function pointer*), jotka voi laittaa osoittamaan tietyn luokan tietynlaisiin jäsenfunktioi-

hin. Sen sijaan C++ ei tunne käsitettä “viite (jäsen)funktioon”, vaikka osoittimet ovatkin mahdollisia.

Listaus 10.11 näyttää esimerkin funktio-osoittimen käytöstä. Siinä on ensin määritelty funktio `onkoAlle5`, joka ottaa kokonaislukuparametrin ja palauttaa totuusarvon, joka kertoo oliko parametri 5:ttä pienempi. Rivillä 10 STL:n algoritmille `find_if` välitetään osoitin tähän funktioon. Algoritmi käy läpi järjestyksessä vektorin alkiot ja kutsuu osoittimen päässä olevaa funktiota antaen kunkin alkion sille parametrina. Algoritmi jatkaa tätä niin kauan, kunnes kaikki alkiot on käyty läpi tai osoittimen päässä oleva funktio on palauttanut arvon tosi. Tällä tavoin `find_if` tässä tapauksessa etsii vektorista ensimmäisen 5:ttä pienemmän alkion.

Samalla tavalla monet muutkin STL:n algoritmit ottavat parametreikseen funktio-osoittimia, joilla alkioita voi testata tai käsitellä. Näin samaa algoritmia voi käyttää useaan eri tarkoitukseen antamalla sille osoitin sopivaan funktioon. Funktio-osoittimien käytössä on kuitenkin rajoituksensa. Jos ohjelmassa haluttaisiin myös etsiä vektorista alkioita, jotka ovat arvoltaan pienempiä kuin *7*, pitäisi ohjelmaan kirjoittaa *uusi* testausfunktio `onkoAlle7`. Tämä ei tietenkään ole järkevä ratkaisu, jos testausfunktioiden määrä kasvaisi suureksi.

Vielä ongelmallisemmaksi tilanne tulee, jos testauksessa käytettävä raja ei olekaan käännösaikana tiedossa, vaan se saadaan funktioon

```
1  bool onkoAlle5(int i)
2  {
3    return i < 5;
4  }
5
6  void tulostaAlle5(vector<int> const& v)
7  {
8    vector<int>::const_iterator i = v.begin();
9
10   while ((i = find_if(i, v.end(), &onkoAlle5)) != v.end())
11   {
12     cout << *i << ' ';
13     ++i;
14   }
15   cout << endl;
16 }
```

LISTAUS 10.11: Funktio-osoittimen välittäminen parametrina

parametrina, kysytään käyttäjältä tai vaikkapa lasketaan ajoaikana. Ongelmana on, että *kaikki* funktion tarvitsemat tiedot täytyy antaa sille parametreina silloin, kun funktiota *kutsutaan* (globaalit muuttujat antavat tähän pienen porsaanreiän, mutta niiden käytön ongelmat ovat yleensä suuremmat kuin hyödyt). Tässä tapauksessa haluttaisiin testauksessa käytettävä raja kiinnittää jo silloin, kun vertailufunktio annetaan `find_if`-algoritmille parametrina, ja `find_if`:n sisällä sitten varsinaisesti kutsuttaisiin funktiota ja annettaisiin sille testattava alkio. C++:n funktiot ja funktio-osoittimet eivät kuitenkaan taivu tällaiseen käyttöön, vaan niiden sijaan täytyy käyttää seuraavassa esiteltäviä funktio-oliota.

10.5.2 Funktio-olioiden periaate

Funktio-oliot (*function object*) ovat olioita, joille on määritelty funktiokutsuoperaattori `()`. Tämän avulla näitä olioita voi “kutsua” aivan kuin ne olisivat funktioita. Verrattuna tavallisiin funktioihin funktio-olioilla on se hyvä puoli, että ne voivat *muistaa* asioita kutsukertojen välillä. Funktio-olioille voi esimerkiksi antaa luomisen yhteydessä tietoja, jotka olio panee talteen. Kun olio sitten välitetään parametrina jollekin algoritmille, se voi kutsujen yhteydessä käyttää näitä sisäänsä talletettuja tietoja hyväkseen.

Joissain C++-teoksissa funktio-olioista käytetään myös nimitystä **funktori** (*functor*). Tätä nimitystä olisi lyhyydestään huolimatta ehkä syytä välttää, koska matematiikassa termi funktori on jo käytössä, ja sillä tarkoitetaan varsin eri asiaa. Funktio-olioilla voidaan sen sijaan saada aikaan samanlaisia vaikutuksia kuin funktionaalisen ohjelmoinnin **sulkeumilla** (*closure*) [Wikström, 1987].

C++:ssa funktio-olioita saadaan aikaan kirjoittamalla luokkia, joissa on määritelty (yksi tai useampi) jäsenfunktio nimeltä **`operator()`**. Listaus 10.12 seuraavalla sivulla näyttää esimerkin tällaisesta. Kun luokasta luodaan olio, voi tätä oliota kutsua ikään kuin se olisi oikea funktio. Syntaksi `olio(parametrit)` aiheuttaa funktiokutsuoperaattorin kutsumisen ikään kuin ohjelmaan olisi kirjoitettu `olio.operator()(parametrit)`.

Hyötynä funktio-olioissa on, että niillä voi normaalien olioiden tapaan olla jäsenmuuttujia, joihin talletetaan tietoja. Tyypillisin tapaus on, että *oliota luotaessa* luokan rakentajalle välitetään parametreja, jotka talletetaan olion jäsenmuuttujiin. Tämän jälkeen oliota kutsut-

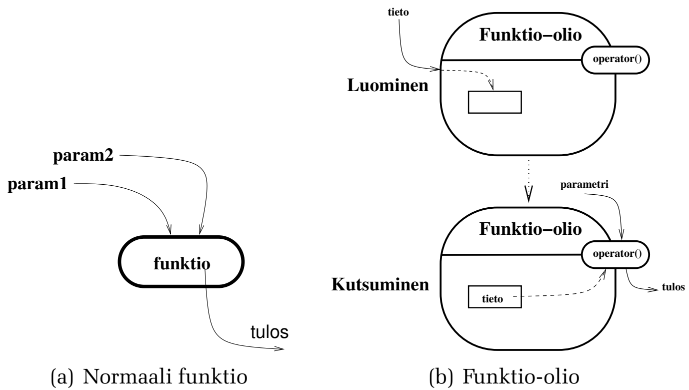
```
1  class OnkoAlle
2  {
3  public:
4    OnkoAlle(int raja);
5    // Funktiokutsuoperaattori
6    inline bool operator()(int verrattava) const;
7  private:
8    int raja_;
9  };
      ⋮
10 OnkoAlle::OnkoAlle(int raja) : raja_(raja)
11 {
12 }
13
14 inline bool OnkoAlle::operator()(int verrattava) const
15 {
16   return verrattava < raja_;
17 }
```

LISTAUS 10.12: Esimerkki funktio-olioluokasta

taessa (sen funktiokutsuoperaattoria kutsuttaessa) olio voi kutsun yhteydessä välitettyjen parametrien lisäksi käyttää hyväkseen myös jäsenmuuttujiin talletettuja tietoja. Periaatteessa olio voi tietysti myös muuttaa jäsenmuuttujiensa arvoja kutsujen yhteydessä, mutta tätä ei yleensä suositellä STL:n yhteydessä muutamaa poikkeusta lukuunottamatta. Kuva 10.6 seuraavalla sivulla havainnollistaa tavallisen funktion ja funktio-olion eroja.

Ohjelmassa funktio-olioita voi käyttää monella eri tavalla. Listauksessa 10.13 seuraavalla sivulla esitellään niistä muutama. Funktiossa `kaytto1` luodaan rivillä 3 funktio-olio aivan tavallisen olion tapaan, ja sille annetaan vertailun rajaksi luku 5. Rivillä 4 funktio-oliota sitten kutsutaan ja sille annetaan vertailtavaksi luvuksi 8. Tämä esimerkki on tarkoitettu vain havainnollistamaan funktio-olioiden syntaksia, sillä näin yksinkertaisessa esimerkissä funktio-oliosta ei vielä ole mitään varsinaista hyötyä.

Listauksen 10.13 toinen esimerkki on jo käyttökelpoisempi. Riveillä 7–13 määritellään funktiomalli `kysyJaTestaa`, joka ottaa *parametrinaan* funktio-olion. Funktiomalli lukee syötteestä luvun ja kutsuu funktio-oliota antaen luetun luvun parametriksi. Jos funktio-olio palauttaa arvon tosi, tulostetaan teksti. Tällainen funktiomalli on var-



(a) Normaali funktio (b) Funktio-olio

KUVA 10.6: Funktio-olioiden idea

```
 1  void kaytto1()
 2  {
 3    OnkoAlle fo(5);
        ⋮
 4    if (fo(8)) { cout << "8 < 5!" << endl; }
 5  }
 6
 7  template <typename FunktioOlio>
 8  void kysyJaTestaa(FunktioOlio const& fo)
 9  {
10    int i;
11    cin >> i; // Virhetarkastelu puuttu
12    if (fo(i)) { cout << "Ehto toteutui!" << endl; }
13  }
14
15  void kaytto2(int raja)
16  {
17    kysyJaTestaa(OnkoAlle(raja));
18    kysyJaTestaa(OnkoAlle(2*raja));
19  }
```

LISTAUS 10.13: Funktio-olion käyttöesimerkkejä

sin yleiskäyttöinen, koska sen ei tarvitse tietää *miten luettua lukua testataan,* koska testaus tapahtuu parametrina saadussa funktio-oliossa.

Lopuksi funktiossa `kaytto2` kutsutaan tätä funktiomallia. Kutsun yhteydessä luodaan suoraan väliaikainen funktio-olio luokan rakentajaa kutsumalla, ja olioon tallentuu testauksen yläraja. Kun kutsu on ohi, tuhotaan väliaikainen funktio-olio automaattisesti.

Funktioiden `kaytto2` ja `kysyJaTestaa` kuvaamaa toiminnallisuutta ei voisi saada aikaan esimerkiksi funktio-osoittimia käyttäen. Listauksessahan `kaytto2` tallettaa *oman parametrinsa* välitettävän funktio-olion *sisään,* ja funktio-oliota puolestaan kutsutaan toisessa funktiossa. Tällaisessa käytössä funktio-oliot ovat lähes välttämättömiä.

Kaikki funktio-osoittimia hyväksyvät STL:n algoritmit hyväksyvät parametreikseen myös funktio-olioita (lisäksi tietyissä tilanteissa STL:ssä funktio-oliot ovat ainoa vaihtoehto). Niinpä aiemmin esitetyn `find_if`-esimerkin voi kirjoittaa yleiskäyttöisemmin myös funktio-olioita käyttämällä. Tämä on tehty listauksessa 10.14, jossa tulostettavien arvojen yläraja saadaan funktioon parametrina.

Funktio-olioiden yhteydessä on syytä huomata, että niitä käyttävät algoritmit saattavat tyypillisesti joskus *kopioida* käyttämiään funktio-olioita. Niinpä jokaisessa funktio-olioluokassa tulisi olla toimiva kopiorakentaja (aliluku 7.1.2). Lisäksi kaikki STL:n algoritmit eivät takaa, että ne käyttäisivät jatkuvasta *samaa kopiota* funktio-oliosta. Näin kaikissa STL:n algoritmeissa funktio-olio ei voi tallettaa sisäänsä kutsukertojen välillä muuttuvaa tietoa, koska seuraava kutsukerta voikin käyttää eri kopiota oliosta. Kaikkein turvallisinta

```
1  void tulostaAlle(vector<int> const& v, int raja)
2  {
3    vector<int>::const_iterator i = v.begin();
4
5    while ((i = find_if(i, v.end(), OnkoAlle(raja))) != v.end())
6    {
7      cout << *i << ' ';
8      ++i;
9    }
10   cout << endl;
11 }
```

LISTAUS 10.14: Funktio-olion käyttö STL:ssä

onkin yleensä tehdä funktiokutsuoperaattorista vakiojäsenfunktio ja pitää funktio-olion jäsenmuuttujien arvot muuttumattomina.

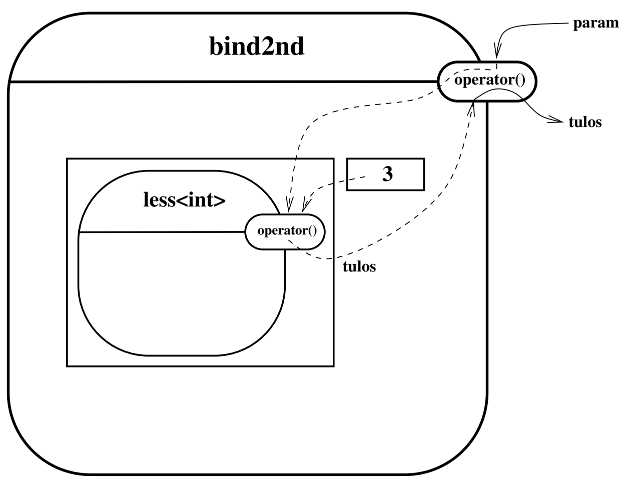
10.5.3 STL:n valmiit funktio-oliot

C++:n standardikirjasto tarjoaa valmiina joukon funktio-olioita, jotta kaikkein tavallisimmissa tapauksissa ohjelmoijan ei aina tarvitsisi kirjoittaa omia funktio-olioluokkiaan. Kaikkia kirjaston funktio-oliotyyppejä ei esitellä tässä, mutta seuraavassa pyritään antamaan suppea yleiskuva siitä, miten kirjaston yleiskäyttöisiä funktio-olioita on tarkoitus käyttää.

Koska on varsin kätevää pystyä välittämään kaikkia C++:n perusoperaatioita myös funktio-olioparametreina, STL määrittelee lähes kaikkia tällaisia operaatioita varten luokkamallit, joista funktio-olioita pystyy luomaan. Tällaisia malleja ovat muun muassa `plus`, `minus`, `multiplies`, `divides`, `equal_to`, `less` ja `greater`. Kaikki nämä ottavat tyyppiparametrinaan funktio-olion parametrien tyypin, joten esimerkiksi kaksi liukulukua yhteenlaskeva funktio-olio olisi `plus<double>` ja kahden kokonaisluvun pienemmyyttä vertailevan funktio-olion tyyppi olisi vastaavasti `less<int>`. Nämä tässä mainitut funktio-oliot eivät sisällä jäsenmuuttujia, vaan ne matkivat aivan tavallisia funktioita ja niillä on vain oletusrakentaja. Tämän vuoksi niitä parametreina välitettäessä funktio-oliot luodaan oletusrakentajaa kutsumalla, siis esimerkiksi `less<int>()`.

Tavallisia funktioita matkivien funktio-olioiden lisäksi STL tarjoaa **funktio-oliosovittimia** (*function object adaptor*), jotka muuntavat olemassa olevia funktio-olioita toisenlaisiksi. Sovittimista tärkeimmät ovat `bind1st` ja `bind2nd`, joiden avulla kaksi parametria ottavista funktio-olioista kumman tahansa parametrin voi "kiinnittää" haluttuun arvoon funktio-olioita luotaessa. Kuva 10.7 seuraavalla sivulla näyttää sovittimen `bind2nd` rakenteen.

Sovittimet ovat itsekin funktio-olioita, joiden funktiokutsuoperaattori ottaa vain yhden parametrin. Jäsenmuuttujissaan sovitin pitää muistissa varsinaisen kaksiparametrisen funktio-olion ja lisäksi kiinnitetyn parametrin arvon. Nämä annetaan sovittimelle sitä luotaessa. Kun sovittimen funktiokutsuoperaattoria kutsutaan yhdellä parametrilla, kutsuu sovitin jäsenmuuttujassaan olevaa funktio-oliota ja antaa sille saamansa parametrin *sekä* jäsenmuuttujaansa talle-



KUVA 10.7: Funktio-olio `bind2nd(less<int>(), 3)`

tetun kiinteän parametrin. Tuloksena saadun paluuarvon sovitin palauttaa itse edelleen kutsujalle.

Funktio-oliosovittimien avulla pystyy muista funktio-olioista muodostamaan kätevästi halutun testauksen tai operaation suoritettavia versioita. Listauksessa 10.15 seuraavalla sivulla näytetään, miten aiemmin esimerkkinä ollut annettua arvoa pienempien alkioiden testaus voidaan koodata STL:n omia funktio-olioita ja sovittimia käyttämällä. Listauksen lauseke `bind2nd(less<int>(), arvo)` luo yhden parametrin ottavan funktio-olion, joka palauttaa toden, jos parametri on pienempi kuin `arvo`. Tämä funktio-olio välitetään sitten parametrina `find_if`-algoritmille.

10.6 C++: Template-metaohjelmointi

Normaalisti tietokoneohjelmia ajettaessa ohjelmat käsittelevät käyttökohteeseensa liittyviä tietoja, jotka niille ajoaikana syötetään (tai jotka on upotettu osaksi itse ohjelman rakennetta). Toisaalta voidaan myös ajatella, että itse tietokoneohjelmakin on vain dataa, jota tietokone käsittelee (suorittamalla sitä). Tämä tulee selkeästi esille siinä,

```
1  #include <functional>
2  using std::less;
3  using std::bind2nd;
4
5  void tulostaAlle2(vector<int> const& v, int raja)
6  {
7    vector<int>::const_iterator i = v.begin();
8
9    while ((i = find_if(i, v.end(), bind2nd(less<int>(),raja))) != v.end())
10   {
11     cout << *i << ' ';
12     ++i;
13   }
14   cout << endl;
15 }
```

LISTAUS 10.15: C++:n funktio-olioiden `less` ja `bind2nd` käyttö

että jokainen C++-ohjelmahan on vain ajoaikaista tietoa C++-kääntäjälle, joka muuntaa C++-koodin konekieliseksi objektitiedostoksi.

10.6.1 Metaohjelmoinnin käsite

Ohjelmia, jotka käsittelevät toisia ohjelmia ja kenties tuottavat lopputuloksenaan uusia ohjelmia, sanotaan **metaohjelmiksi**[☺] (*metaprogram*). Tällaisia ohjelmia ovat varsinaisten kääntäjien lisäksi myös erilaiset ohjelmageneraattorit, esikääntäjät, käyttöliittymägeneraattorit ja niin edelleen. Vastaavasti metaohjelmien kirjoittamista kutsutaan yleisesti **metaohjelmoinniksi** (*metaprogramming*). Kirjan "Generative Programming" [Czarnecki ja Eisenecker, 2000] luvussa 10 on varsin hyva yleiskatsaus metaohjelmoinnin käsitteisiin.

Metaohjelmointi muuttuu huomattavasti mielenkiintoisemmaksi, jos ohjelma pystyy tutkimaan *omaa* rakennettaan ja kenties vaikuttamaan omaan koodiinsa. Tällaista ohjelman kykyä "itsetutkiskeluun" kutsutaan nimellä **reflektio** (*reflection*). Joissain kielissä kuten Smalltalk:ssa tuki reflektiolle on varsin laaja, ja Smalltalk-ohjelmat pystyvät tutkimaan omaa luokkahierarkiaansa, luomaan uusia aliluokkia

[☺] Kreikan sana "meta" tarkoittaa suunnilleen samaa kuin "jälkeen". Sitä käytetään yleisesti etuliitteenä, jolla kuvataan alkuperäisen käsitteen suhteen "ylemmällä tasolla" olevaa asiaa. Niinpä metaohjelmointi on ohjelmointia, jossa metaohjelma käsittelee varsinaista ohjelmakoodia.

tai jopa ajoaikana muuttamaan olion tyypin toiseksi. Aliluvun 8.2.1 metaluokat ovat yksi esimerkki Smalltalk:n reflektio-ominaisuuksista.

C++:n ja Javan tapaisissa käännettävissä kielissä mahdollisuus reflektioon on yleensä paljon rajoitetumpi. Javan `Reflection`-rajapinnan avulla ohjelma voi jonkin verran tarkastella omaa rakennettaan ja esim. kysyä ajoaikana, mitä jäsenfunktioita luokasta löytyy ja vaikkapa kutsua jäsenfunktiota, jonka nimi löytyy merkkijonomuuttujasta. Sen sijaan helppoa mahdollisuutta ohjelmankoodin muuttamiseen tai tuottamiseen ei ole. C++:n mahdollisuudet reflektioon ajoaikana ovat vielä rajoitetummat. Aliluvussa 6.5.3 esitellyt **`dynamic_cast`** ja **`typeid`** ovat rajoitettuja yksinkertaisia esimerkkejä tilanteista, joissa ohjelma pystyy tutkimaan omaan rakenteeseensa liittyviä asioita (tässä olion sijaintia luokkahierarkiassa). Samalla tavoin **`sizeof`**-operaattorin avulla ohjelma voi kysyä, montako tavua muistia jonkin tyyppi vie.

Vaikka C++:n ajoaikaiset metaohjelmointimahdollisuudet ovat varsin rajoitetut, tekevät kielen funktio- ja luokkamallit mahdolliseksi **käännösaikaisen metaohjelmoinnin** (*static metaprogramming*), jota C++:ssa usein kutsutaan myös **template-metaohjelmoinniksi** (*template metaprogramming*). Siinä template-mekanismin avulla voidaan kirjoittaa metaohjelmia, jotka tutkivat tyyppiparametreina annettuja tyyppejä ja vakioita ja tekevät niiden perusteella päätöksiä tuotettavasta koodista. Tätä mahdollisuutta ei missään vaiheessa suunniteltu tarkoituksellisesti C++-kieleen, vaan se havaittiin “vahingossa” 90-luvun puolivälissä. Tämän vuoksi C++:n template-metaohjelmointi on käyttökelpoisuudestaan huolimatta usein työlästä ja kirjoitettu koodi vaikealukuista.

Käännösaikaista metaohjelmointia voi ajatella myös niin, että kirjoitettu ohjelma jakautuu ikään kuin kahteen osaan: käännösaikana suoritettavaan metakoodiin, joka voi tutkia rajoitetusti ohjelman rakennetta ja vaikuttaa sen perusteella käännettävään C++-koodiin. Tämä käännetty koodi suoritetaan sitten ajoaikana, ja siinä kaikki “metatason” asiat on jo kiinnitetty eikä niihin enää voi vaikuttaa. Seuraavissa aliluvuissa tutustutaan lyhyesti C++:n template-metaohjelmoinnin mahdollisuuksiin.

10.6.2 Metaohjelmointi ja geneerisyys

Perinteistä ohjelmaa kirjoitettaessa tarvetta ohjelman "itsetutkiskeluun" ja reflektioon harvemmin esiintyy, mutta tilanne muuttuu nopeasti, kun aletaan suunnitella yleiskäyttöisiä ohjelmakomponentteja, joissa (kuten aliluvussa 9.1 todettiin) on yleensä sekä samanlaisena pysyviä osia että käyttökohteesta riippuvaa koodia. C++:n template-mekanismi antaa tähän erotteluun mahdollisuuden, kun itse funktio- tai luokkamalli edustaa pysyvää yleiskäyttöistä koodia ja auki jätetyt tyyppiparametrit käyttökohteen mukaan määrättäviä asioita.

Varsin helposti törmätään tilanteeseen, jossa pelkkä auki jätetyn tyypin käyttäminen yleiskäyttöisessä template-koodissa ei riitä, vaan koodin pitäisi pystyä tutkimaan auki jätettyjen tyyppien ominaisuuksia ja tehdä niiden perusteella päätöksiä esim. toisten tyyppien tai käytettävän algoritmin valinnasta. C++:n tapauksessa metaohjelmointi tapahtuu lähes yksinomaan template-mekanismin avulla, ja lisäksi sitä tyypillisesti käytetään nimenomaan funktio- ja luokkamallien avulla toteutetuissa yleiskäyttöisissä ohjelmakomponenteissa.

C++:ssa ja muissa suoraan konekielelle käännettävissä kielissä metaohjelmointia rajoittaa se, että lopullinen konekielinen ohjelmabinaari on kiinteä, eikä siinä enää voi olla auki jätettynä ohjelman rakenteeseen liittyviä asioita. Samasta syystä kaikki funktio- ja luokkamallitkin instantioidaan jo käännösaikana. Niinpä C++:n template-metaohjelmointi rajoittuukin käännösaikaisiin päätöksiin käännettävän ohjelman rakenteesta. Tilannetta on ehkä helpointa ajatella niin, että metaohjelmoinnilla kirjoitettava "metakoodi" suoritetaan jo ohjelmaa käännettäessä, ja se vaikuttaa ainoastaan siihen, millainen lopullisesta ohjelmabinaarista tulee.

Tyypillinen käyttökohde metaohjelmoinnille yleiskäyttöisissä ohjelmakirjastoissa on optimointi. Metaohjelmoinnin avulla yleiskäyttöinen kirjasto voi käännösaikana valita käyttökohteeseen sopivan algoritmin tai säätää ja virittää algoritmia kohteeseen parhaiten sopivaksi. Tällöin puhutaan usein **mukautuvasta järjestelmästä** (*adaptive system*). Yksinkertainen esimerkki tästä on aliluvussa 10.2.1 `vector`-luokkamallin yhteydessä mainittu `vector<bool>`, jossa STL valitsee `bool`-tyypin tapauksessa vektorille muistinkulutuksen kannalta tehokkaamman toteutuksen. Käännösaikainen metaohjelmointi tekee kuitenkin mahdolliseksi myös monimutkaisemman mukautumisen.

10.6.3 Metafunktiot

Perinteisessä ohjelmoinnissa funktiot ovat ohjelman perusosia. Mahdollisimman abstraktilla tasolla ajatellen funktiot ovat ohjelman rakenteita, jotka ajoaikana tuottavat niille annettujen parametrien perusteella paluuarvon (paluuarvon lisäksi funktioilla voi olla myös sivuvaikutuksia, mutta niistä ei tässä välitetä). Samalla tavoin **metafunktioilla** (*metafunction*) tarkoitetaan metaohjelman rakenteita, jotka tuottavat niille annetun ohjelman rakenteeseen liittyvän tiedon (esim. tyypin) perusteella jotain muuta ohjelman rakenteeseen liittyvää. C++:n käännösaikaisessa metaohjelmoinnissa on monia eri tapoja saada aikaan metafunktioina toimivia rakenteita. Tyypillisesti niiden parametreina on ohjelman tyyppejä tai käännösaikaisia vakioita, joiden perusteella metafunktio tuottaa toisia tyyppejä, valitsee tuotettavaa koodia tai laskee uusia käännösaikaisia vakioita. On huomattava, että funktiomallit ja metafunktiot *eivät* ole alkuunkaan sama asia, vaikka tällaista väärinkäsitystä joskus esiintyy ja vaikka funktiomalleja voidaan metaohjelmointiin käyttääkin.

Ehkä yksinkertaisin esimerkki C++:n metafunktiosta on kielen sisäänrakennettu operaattori **`sizeof`**. Sille annetaan parametrina jokin kielen tyyppi, ja **`sizeof`** laskee käännösaikana, montako tavua muistia kyseisen tyyppinen muuttuja tarvitsisi.[Ω] Esimerkiksi **`sizeof(int)`** palauttaa ohjelmaa käännettäessä arvon `4`, jos **`int`**-tyyppi vaatii kyseisessä ympäristössä 4 tavua muistia. Olennaista **`sizeof`**-operaattorissa on, ettei siitä aiheudu minkäänlaista ajoaikaista suoritettavaa koodia, vaan koko operaatio suoritetaan jo ohjelmaa käännettäessä.

Toinen jopa hämäävän yksinkertainen tapa saada aikaan metafunktioita ovat luokkien sisällä määritellyt vakiot ja tyypit. Oletetaan, että meillä on ohjelmassa STL:n säiliö `vector<int>`, johon ohjelmakoodissa halutaan iteraattori. Kuten aiemmin on todettu, löytyy iteraattorin tyyppi säiliön sisäisenä tyyppinä syntaksilla `vector<int>::iterator`. Tarkemmin ajatellen tässäkin on kyse metafunktiosta — säiliön tyypin perusteella saadaan käännösaikana selville säiliöön sopivan iteraattorin tyyppi. Samalla tavoin säiliön alkiotyypin saa selville metafunktiolla `::value_type`. Selkeämmin me-

[Ω]Vaihtoehtoisesti **`sizeof`**-operaattorille voi antaa parametriksi minkä tahansa C++:n lausekkeen. Tällöin **`sizeof`** kertoo, kuinka monta tavua muistia lausekkeen lopputulos vaatii. Itse lausekkeen arvoa *ei* kuitenkaan lasketa missään vaiheessa, muistinkulutuksen kun voi päätellä jo itse lausekkeen rakenteesta.

tataso näkyy listauksen 10.16 funktiomallista, joka etsii annetusta säiliöstä pienimmän alkion ja palauttaa sen. Sekä funktiomallin paluuarvon että iteraattoreiden tyypin päätteleminen säiliön tyypin perusteella on yksinkertaista metaohjelmointia.

Luokkien sisälle upotetut tyypit kuten `::value_type` ja `::iterator` eivät ole kovin yleiskäyttöinen tapa metafunktioiden kirjoittamiseen, koska niiden käyttö vaatii, että *kaikki* luokat määrittelevät sisällään samalla tavoin nimetyt tyypit. Uuden "metafunktion" lisääminen vaatii kaikkien metafunktion parametrina käytettävien luokkien päivittämisen niin, että niiden sisään lisätään haluttu uusi tyyppimääritys. Tämä ei tietenkään ole mahdollista, jos käytössä ei ole näiden luokkien lähdekoodia. Esimerkiksi STL:n säiliötyyppeihin ei ohjelmoija voi itse lisätä omia tyyppimäärityksiään, vaikka tähän tulisikin tarvetta.

C++:n luokkamallit ja niihin liittyvä mahdollisuus mallien erikoistamiseen antavat kuitenkin mahdollisuuden kirjoittaa "irrallisia" metafunktioita, jotka tuottavat mallin parametrien perusteella uusia tyyppejä tai käännösaikaisia vakioita. Tämä tapahtuu kirjoittamalla erillinen luokkamalli, jonka sisälle on määritelty sisäisiä tyyppejä, joiden arvo riippuu tyyppiparametrista. Tällaisista malleista käytetään englanniksi usein nimitystä *trait*. Listaus 10.17 seuraavalla sivulla näyttää yksinkertaisen trait-metafunktion `ToistoStruct`, joka laskee **`struct`**-tietorakenteita, joissa on useita annetun tyyppisiä kent-

```
1  template <typename Sailio>
2  typename Sailio::value_type pienin(Sailio const& s)
3  { // Esimerkin lyhentämiseksi virhetarkastelu puuttuu
4    typename Sailio::const_iterator iter = s.begin();
5    typename Sailio::const_iterator pieninIter = iter;
6
7    while (iter != s.end())
8    {
9      if (*iter < *pieninIter) { pieninIter = iter; }
10     ++iter;
11   }
12
13   return *pieninIter;
14 }
```

LISTAUS 10.16: Esimerkki yksinkertaisesta metaohjelmoinnista

tiä. Esimerkiksi kolme **int**-kenttää sisältävä **struct** saataisiin syntaksilla `ToistoStruct<int>::Kolmikko`. Koska tällaisten trait-mallien ainoa tarkoitus on tarjota sisäisiä tyyppimäärittelyjä, näkee C++:ssa usein käytettävän **struct**-avainsanaa **class**:n sijaan, koska kyseessä ei varsinaisesti ole "luokka", josta tehtäisiin olioita, ja koska **struct**-rakenteessa oletusnäkyvyys on **public** eikä **private**.

Äskeisen esimerkin mukaiset metafunktiot antavat mahdollisuuden ainoastaan yksinkertaiseen uusien tyyppien luomiseen. Luokkamallien erikoistus ja osittaiserikoistus antavat kuitenkin mahdollisuuden huomattavasti monipuolisempiin ja käyttökelpoisempiin metafunktioihin. Niiden avulla on template-metaohjelmoinnissa mahdollista saada aikaan if-lausetta vastaavia ehtorakenteita.

Vastaavasti mallien rekursiivinen instantioiminen antaa mahdollisuuden toistorakenteisiin. Näiden kahden ominaisuuden avulla periaatteessa minkä tahansa algoritmin saa koodattua käännösaikana suoritettavaksi template-metaohjelmaksi. Käytännössä tällaisten ohjelmien koodi on kuitenkin helposti vaikealukuista, koska templatemekanismia ei koskaan ole suunniteltuun varsinaiseen metaohjel-

```
1  template <typename Tyyppi>
2  struct ToistoStruct
3  {
4    struct Yksikko
5    {
6      Tyyppi eka;
7    };
8    struct Kaksikko
9    {
10     Tyyppi eka;
11     Tyyppi toka;
12   };
13   struct Kolmikko
14   {
15     Tyyppi eka;
16     Tyyppi toka;
17     Tyyppi kolmas;
18   };
        ⋮
19 };
```

LISTAUS 10.17: Yksinkertainen trait-metafunktio

mointiin.

Listaus 10.18 näyttää esimerkin erikoistamalla tehdystä metafunktiosta. Metafunktio `IntKorotus` ottaa parametrikseen kaksi kokonaislukutyyppiä, ja sen sisäinen “paluuarvotyyppi” `Tulos` kertoo, mikä kokonaislukutyyppi pystyy pitämään sisällään kummankin annetun kokonaislukutyypin kaikki mahdolliset arvot. Tätä metafunktiota käytetään listauksessa päättelemään käännösaikana `min`-funktiomallin paluutyyppi, kun `min`:n parametrit voivat olla keskenään erityyppisiä. `IntKorotus` on toteutettu niin, että itse “perusversio” mallista on tyhjä, ja kaikki päättely on koodattu erikoistuksiin, joita pitäisi

```
1  template <typename T1, typename T2>
2  struct IntKorotus
3  {
4    // Erikoistamaton versio on tyhjä, ei osata tehdä mitään
5  };
6
7  template <>
8  struct IntKorotus<char, int>
9  {
10   typedef int Tulos;
11 };
12
13 template <>
14 struct IntKorotus<short int, char>
15 {
16   typedef short int Tulos;
17 };
18
19 template <>
20 struct IntKorotus<short int, unsigned short int>
21 {
22   typedef int Tulos; // Kääntäjäriippuvaista, riittääkö int
23 };
       ⋮
24 template <typename T1, typename T2>
25 typename IntKorotus<T1, T2>::Tulos min(T1 p1, T2 p2)
26 {
27   if (p1 < p2) { return p1; }
28   else { return p2; }
29 }
```

LISTAUS 10.18: Erikoistamalla tehty template-metafunktio

kirjoittaa koodiin yksi jokaista mahdollista kokonaislukutyyppiparia varten.

Metafunktio sisältää myös virhetarkastelun siinä mielessä, että jos `IntKorotus`-mallin parametrit eivät ole kokonaislukutyyppejä, valitsee kääntäjä mallin erikoistamattoman perustoteutuksen, joka ei määrittele tyyppiä `Tulos` ollenkaan. Tämä puolestaan aiheuttaa käännösvirheen, kun tyyppiin `Tulos` viitataan.

Luokkamallien osittaiserikoistus antaa mahdollisuuden hienovaraisempiinkin metafunktioihin. Listauksen 10.19 metafunktio poistaa vakiomääreen **`const`** tyyppiparametristaan. Jos parametrina annettu tyyppi ei ole **`const`**, valitaan erikoistamaton versio, jossa on `Tulos` sama kuin tyyppiparametri. Jos sen sijaan tyyppiparametri on vakiotyyppiä, valitaan mallin osittaiserikoistus. Siinä tyyppiparametriin `T` jääkin enää vain alkuperäinen ei-vakiotyyppi, koska osittaiserikoistuksen tyyppimääreenä oleva <`T` **`const`**> tekee selväksi, että `T` viittaa tyypin “ei-**`const`**”-osaan. Listauksessa käytetään tätä metafunktiota pitämään huoli siitä, että listauksen `min`-toteutuksen muuttuja `pienin` ei ole vakio, jotta siihen voi sijoittaa.

```
 1 template <typename T>
 2 struct PoistaConst // Oletus, kun tyyppi on "normaalityyppi"
 3 {
 4   typedef T Tulos;
 5 };
 6
 7 template <typename T>
 8 struct PoistaConst<T const> // Osittaiserikoistus kaikille vakiotyypeille
 9 { // Huomaa, että tässä T on nyt ilman const-määrettä oleva tyyppi
10   typedef T Tulos;
11 };
       ⋮
12 template <typename T>
13 T min(T* p1, T* p2)
14 {
15   typename PoistaConst<T>::Tulos pienin;
16   if (*p1 < *p2) { pienin = *p1; } // Sijoitus, joten pienin ei saa olla vakio
17   else { pienin = *p2; }
18   return pienin;
19 }
```

LISTAUS 10.19: Osittaiserikoistuksella tehty metafunktio

Edellä mainittuja tekniikoita käyttäen on mahdollista kirjoittaa metafunktiokirjastoja, jotka helpottavat varsinkin luokka- ja funktiomallien avulla tehtyä geneeristä ohjelmointia. Tällaisia kirjastoja löytyy myös valmiina. Esimerkiksi kirjassa "Modern C++ Design" [Alexandrescu, 2001] esitelty kirjasto Loki tarjoaa valmiita metafunktiototeutuksia joihinkin suunnittelumalleihin. Samoin C++-kirjastokokoelma Boost [Boost, 2003] sisältää monia käyttökelpoisia metafunktioita, joista osa saattaa päätyä seuraaviin C++-standardin versioihin.

10.6.4 Esimerkki metafunktioista: `numeric_limits`

Template-metaohjelmointi on yleistynyt C++-ohjelmoijien parissa vasta viime vuosina. Siitä huolimatta jo nykyisessä C++-standardissa on jonkin verran metafunktioiksi laskettavia ominaisuuksia. Tässä aliluvussa esitellään standardin metafunktio `numeric_limits`, joka on hyvä esimerkki siitä, miten metafunktioiden avulla voidaan saada hyödyllistä tietoa ohjelman numeerisista tyypeistä.

Silloin tällöin ohjelmoinnissa tulee tarve saada tietoa esimerkiksi siitä, mikä on suurin mahdollinen **`int`**-tyyppiin mahtuva luku tai kuinka monen numeron tarkkuudella **`double`** luvut esittää. C-kielessä ja standardia edeltävässä C++:ssa tämä ratkaistiin niin, että nämä tiedot löytyivät sopivan includen jälkeen vakioina. Edelleenkin C++:ssa voi lukea otsikkotiedoston `<climits>`, jonka jälkeen suurin **`int`** löytyy nimellä `INT_MAX` ja otsikkotiedoston `<cfloat>` lukemisen jälkeen **`double`**-tyypin tarkkuus numeroina selviää vakiosta `DBL_MANT_DIG`.

Tämä vakioihin perustuva järjestelmä ei kuitenkaan ole käyttökelpoinen geneerisessä ohjelmoinnissa. Tyyppiparametrin `T` perusteella ei millään voi selvittää, mikä kyseisen tyypin maksimiarvo on, koska tässä C:stä periytyvässä menetelmässä jokaisen tyypin tiedot löytyvät *eri nimisistä* vakioista, eikä tyyppiparametrin perusteella pysty millään käännösaikana muodostamaan vakiolle oikeaa nimeä. Tämän vuoksi C++-standardiin lisättiin trait-metafunktio `numeric_limits`, joka kertoo tyyppiparametrinsa perusteella kyseiseen tyyppiin liittyviä tietoja.

Metafunktio saadaan käyttöön lukemalla otsikkotiedosto `<limits>`. Tämän jälkeen ohjelma voi kysyä minkä tahansa numeerisen tyypin tietoja syntaksilla `numeric_limits<tyyppi>::tieto`. Esimerkiksi **`int`** tyypin maksimiarvo on `numeric_limits<int>::max()`.

Samoin tyypin **double** tarkkuus 10-järjestelmässä on numeric_limits<**double**>::digits10 numeroa.

Vaikka tämä uusi tapa on syntaksiltaan pidempi kuin vanha, sen etuna on, että esimerkiksi template-koodin sisällä tyyppiparametrin T pienin mahdollinen arvo selviää kutsulla numeric_limits<T>::min() tyypistä T riippumatta. Taulukko 10.8 luettelee tärkeimmät numeric_limits-metafunktion tarjoamat tiedot. Tarkemmat selitykset metafunktion toiminnasta ja yksityiskohdista saa monista C++-oppaista.

numeric_limits<T>::	**Selitys**
min(), max()	Tyypin pienin ja suurin arvo. Liukulukutyypeillä min() kertoo *pienimmän positiivisen arvon*.
radix	Tyypin sisäinen lukujärjestelmä (yleensä 2).
digits, digits10	Tyypin tarkkuus numeroina radix-kantaisessa esitysmuodossa (digits) ja 10-järjestelmässä (digits10).
is_signed, is_integer, is_exact	Totuusarvo **true** tai **false** riippuen siitä, onko tyyppi etumerkillinen, kokonaislukutyyppi ja tarkka (siis pyöristysvirheitä ei voi sattua).
is_bounded	Onko lukualue rajoitettu. Tosi kaikille sisäänrakennetuille tyypeille, mutta omille rajoittamattoman tarkkuuden tyypeille voisi olla epätosi.
is_modulo	Onko tyyppi modulo-aritmetiikkaa käyttävä, eli pyörähtääkö lukualue "ympäri". Tosi etumerkittömille kokonaisluvuille ja useimmiten muillekin kokonaisluvuille. Yleensä epätosi liukulukutyypeille.
epsilon(), round_error()	Liukulukutyyppien tarkkuuteen liittyviä tietoja. Tarkoilla luvuilla 0.
min/max_exponent, min/max_exponent10	Liukulukutyyppien eksponentin minimi- ja maksimiarvot sisäisen lukujärjestelmän että 10-järjestelmän eksponenteille. Kokonaislukutyypeillä aina 0.
is_iec559, has_infinity, has_denorm ...	IEC-559-standardin mukaisten liukulukujen tietoja, yksityiskohtainen selittäminen veisi liikaa tilaa.☺

KUVA 10.8: Metafunktion numeric_limits palvelut

Sen lisäksi, että numeric_limits kertoo yhtenäisellä tavalla kaikista C++:n sisäisistä tyypeistä, se on myös laajennettavissa omille lukuja esittäville luokille. Toteutukseltaan numeric_limits on nimittäin jokaiselle perustyypille erikoistettu **struct**-luokkamalli. Niinpä sitä voi edelleen erikoistaa omille lukutyypeille. Jos ohjelma esimerkiksi määrittelee oman murtolukutyypin, voi sitä varten kirjoittaa erikoistuksen numeric_limits-mallille, jolloin numeric_limits:ä käyttävät geneeriset laskenta-algoritmit toimivat myös tälle tyypille.

Listaus 10.20 näyttää funktiomallin, joka vertailee annettujen lukujen yhtäsuuruutta. Kokonaisluvuille ja muille tarkoille luvuille se käyttää ==-vertailua, liukuluvuille ja muille pyöristysvirheistä kärsiville luvuille erotuksen itseisarvon suuruusluokan vertaamista mahdolliseen pyöristysvirheeseen.

```
1  #include <limits>
2  using std::numeric_limits;
3  #include <cmath>
4  using std::abs; // Itseisarvo
5  #include <algorithm>
6  using std::max;
7
8  template <typename T>
9  bool yhtasuuruus(T p1, T p2)
10 {
11    if (numeric_limits<T>::is_exact)
12    { // Pyöristysvirheet eivät ongelma, vertaillaan suoraan
13      return p1 == p2;
14    }
15    else
16    { // Pyöristysvirhemahdollisuus, käytetään kaavaa |p1−p2| / max(|p1|,|p2|) ≤ ε
17      return
18        abs(p1-p2) / max(abs(p1),abs(p2)) <= numeric_limits<T>::epsilon();
19      // Tarkasti ottaen tämä kaava ei riitä aina, mutta kelvatkoon.☺
20    }
21 }
```

LISTAUS 10.20: Esimerkki numeric_limits-metafunktion käytöstä

10.6.5 Esimerkki metaohjelmoinnista: tyypin valinta

Edellä esitetyt tavat metafunktioiden kirjoittamiseen tekevät metaohjelmoinnin mahdolliseksi, mutta metaohjelmien kirjoittaminen on varsin työlästä. Yksinkertaisenkin valinnan suorittaminen käännösaikana vaatii erillisen luokkamallin ja sen erikoistuksen kirjoittamista. Olisikin kätevää, jos C++:n metakoodiin saataisiin edes etäisesti normaaleja ohjelmointikielen rakenteita muistuttavia mekanismeja.

Tällaisia valinta-, silmukka- ynnä muita rakenteita on onneksi mahdollista kirjoittaa erillisinä metafunktioina, joita voi kutsua metakoodista. Tämä tekee metaohjelmista jonkin verran helppolukuisempia, vaikka edelleenkään template-metakoodi ei yllä selkeydessä läheskään normaalin ohjelmakoodin tasolle. Tässä aliluvussa esitetään esimerkkinä yksinkertaisen ehtolauseen `IF` toteuttaminen metafunktiona. Vastaavalla tavalla voidaan toteuttaa toistolauseet kuten `WHILE` ja muut kontrollirakenteet, mutta niistä kiinnostuneen kannattaa tutustua esimerkiksi kirjan "Generative Programming" [Czarnecki ja Eisenecker, 2000] lukuun 10 "Static Metaprogramming in C++".

Toteutettavan `IF`-metafunktion ideana on ottaa template-parametrina yksi **`bool`**-tyyppinen käännösaikainen vakio, joka edustaa testattavaa ehtoa, ja kaksi tyyppiä. `IF` tuottaa paluuarvokseen jommankumman näistä tyypeistä riippuen siitä onko ehto tosi vai epätosi. Itse metafunktion toteutus on kohtalaisen yksinkertainen ja perustuu luokkamallin osittaiserikoistukseen.

Metafunktion koodi on listauksessa 10.21 seuraavalla sivulla. Sen erikoistamaton perustoteutus vastaa tilannetta, jossa ehto on tosi, joten perustoteutuksessa sisäinen tyyppi `Tulos` määritellään samaksi kuin ensimmäinen tyyppiparametri `Tosi`. Lisäksi mallille määritellään osittaiserikoistus sitä tapausta varten, että ehto on epätosi. Tällöin tyyppi `Tulos` määritellään samaksi kuin toinen tyyppiparametri `Epatosi`. Lopputuloksena on metafunktio, jossa `Tulos` on joko sama tyyppi kuin `Tosi` tai `Epatosi` totuusarvosta riippuen.

Toteutetun metafunktion avulla voidaan nyt kirjoittaa koodia, jossa käytetty tyyppi valitaan käännösaikana annetun ehdon perusteella. Esimerkiksi halutaan valita muuttujan tyypiksi **`int`**, jos tämä tyyppi on vähintään neljän tavun kokoinen, muuten valitaan **`long int`**. Tämä onnistuu seuraavasti:

```
IF< (sizeof(int)>=4), int, long int >::Tulos muuttuja = 0;
```

```
4  // Perustapaus, jos ehto on tosi
5  template <bool EHTO, typename Tosi, typename Epatosi>
6  struct IF
7  {
8    typedef Tosi Tulos;
9  };
10
11 // Osittaiserikoistus, jos ehto on epätosi
12 template <typename Tosi, typename Epatosi>
13 struct IF<false, Tosi, Epatosi>
14 {
15   typedef Epatosi Tulos;
16 };
```

LISTAUS 10.21: Metafunktion `IF` toteutus

Listaus 10.22 näyttää toisen käyttöesimerkin metafunktiolle `IF`. Kyseessä on sama `min`-funktiomallin paluutyypin valinta kuin listauksessa 10.18, mutta nyt erillisen paluutyyppiä varten kirjoitetun metafunktion sijaan valitaan kahdesta parametrityypistä se, joka koko numeroina on `numeric_limits`:n mukaan suurempi (tämä versio ei ota huomioon esimerkiksi parametrityyppien etumerkkejä vaan olettaa että molemmat parametrit ovat joko etumerkillisiä tai etumerkittömiä kokonaislukuja).

```
1  // Valitaan paluutyypiksi parametreista "laajempi"
2  template <typename T1, typename T2>
3  typename IF< (std::numeric_limits<T1>::digits >=
4                std::numeric_limits<T2>::digits),
5                T1, T2 >::Tulos
6  min(T1 const& p1, T2 const& p2)
7  {
8    if (p1 < p2)
9    {
10     return p1;
11   }
12   else
13   {
14     return p2;
15   }
16 }
```

LISTAUS 10.22: Metafunktion `IF` käyttöesimerkki

10.6.6 Esimerkki metaohjelmoinnista: optimointi

Tähän mennessä kaikki esitetyt metafunktiot ovat vain tuottaneet uusia tyyppejä tai valinneet käännösaikana olemassa olevien tyyppien välillä. Toinen tärkeä käyttökohde metaohjelmoinnille on koodin optimointi käännösaikana tehtävien päätelmien perusteella. Käytännössä tämä tarkoittaa usein, että useista eri algoritmivaihtoehdoista valitaan metaohjelmoinnin avulla tehokkain käyttökohteeseen sopiva, tai algoritmin toimintaa säädetään käännösaikaisten laskelmien perusteella. Tällaisesta optimoinnista on usein suuri hyöty, koska se voidaan tehdä huomattavasti korkeammalla abstraktiotasolla kuin mihin C++-kääntäjien omat optimointialgoritmit pystyvät.

Esimerkkinä tämantapaisesta korkean tason optimoinnista käytetään tässä STL:n iteraattoreita. Kuten aiemmin on todettu, iteraattoreita pystyy liikuttelemaan säiliön sisällä eri tavoin riippuen siitä, mihin iteraattorikategoriaan ne kuuluvat. Eteenpäiniteraattorit pystyvät liikkumaan vain yhden alkion eteenpäin kerrallaan. Kaksisuuntaiset iteraattorit taas pystyvät liikkumaan yhden alkion verran eteen- tai taaksepäin. Hajasaanti-iteraattorit puolestaan pystyvät hyppimään mielivaltaisen hypyn kumpaan suuntaan tahansa. Jaottelu kategorioihin STL:ssä perustui siihen, että kaikkien iteraattoreiden liikkumisien taataan tapahtuvan vakioajassa.

Joissain tapauksissa olisi tärkeää pystyä siirtämään iteraattoria halutun verran eteen- tai taaksepäin riippumatta siitä, kuinka tehokasta liikkuminen on. Tämä on tietysti mahdollista käyttämällä kaikille iteraattoreille käyttämällä ++-operaattoria silmukassa, mutta silloin siirtäminen tulee tehtyä turhan tehottomasti hajasaanti-iteraattoreille, jotka pystyisivät hyppäämään halutun verran kerrallakin. Olisi ihanteellista, jos *sama siirtymisfunktio* pystyisi siirtämään hajasaanti-iteraattoreita +=-operaatiolla ja muita iteraattoreita silmukassa ++-operaattorilla.

Ilman metaohjelmointia tätä ongelmaa ei ole mahdollista ratkaista helposti C++:ssa. Pelkkä normaali `if`-lause ei riitä, koska muille kuin hajasaanti-iteraattoreille +=-operaattoria käyttävä koodi *aiheuttaa käännösvirheen.* Täten sama C++-funktio ei pysty ajoaikana päättelemään miten iteraattoria tulisi liikuttaa tehokkaasti.

Ratkaisuna ongelmaan on kirjoittaa kuormitettu funktiomalli, jolla on eri toteutukset eri iteraattorityyppejä varten. Näistä toteutuksista valitaan sitten käännösaikana kutakin kutsua varten sopiva. Tätä

varten täytyy kuitenkin pystyä käännösaikana selvittämään *mihin iteraattorikategoriaan* tietty iteraattori kuuluu. Onneksi STL tarjoaa juuri tätä tarkoitusta varten valmiin trait-metafunktion `iterator_traits`.

Metafunktiolle `iterator_traits` annetaan tyyppiparametrina iteraattori (tai C++:n perustaulukkotyypin tapauksessa normaali osoitin, joka toimii iteraattorina taulukkoon). Tämän jälkeen metafunktio kertoo iteraattorin ominaisuuksia. Tätä esimerkkiä varten meitä kiinnostaa ominaisuuksista vain tyyppi `iterator_traits<T>::iterator_category`. Tämä tyyppi on jokin tyypeistä `input_iterator_tag`, `output_iterator_tag`, `forward_iterator_tag`, `bidirectional_iterator_tag` tai `random_access_iterator_tag`. Nämä tyypit on periytetty toisistaan niin, että saadaan iteraattorikategorioita vastaava luokkahierarkia, joka näkyi esimerkiksi kuvassa 10.5 sivulla 330.

Tätä iteraattorikategorioihin jakamista voidaan nyt käyttää hyväksi algoritmin valinnassa. Listauksessa 10.23 seuraavalla sivulla on kolme eri toteutusta funktiomallille `siirry_apu`.Nämä eroavat toisistaan viimeisen parametrin tyypin osalta. Tätä viimeistä parametria käytetään iteraattorikategorian valintaan.[♍] Ensimmäinen versio funktiomallista hyväksyy kolmanneksi parametriksi lukuiteraattorikategorian (johon kuuluvat periyttämällä kaikki iteraattorikategoriat tulostusiteraattoria lukuun ottamatta). Toisen version lisäparametrin tyyppi määrää kategoriaksi kaksisuuntaisen iteraattorin ja kolmas hajasaanti-iteraattorin. Nämä eri versiot on toteutettu eri tavoilla niin, että kutakin iteraattorityyppiä liikutetaan niin tehokkaasti kuin mahdollista.

Varsinainen funktiomalli `siirry` on toteutettu riveillä 43–48 niin, että se kutsuu apufunktiota `siirry_apu` ja antaa sille kolmanneksi parametriksi `iterator_traits`-metafunktion avulla saadun iteraattorikategorian. Tämän kategorian perusteella kääntäjä sitten kutsuu kyseiselle iteraattorille sopivaa apufunktiota, joka siirtää iteraattorin tehokkaimmalla tavalla halutun verran eteen- tai taaksepäin.

Kuten tästä esimerkistä huomataan, metaohjelmointi antaa C++:ssa varsin monipuoliset mahdollisuudet koodin korkealla tasolla tapahtuvaan optimointiin. Ikävä kyllä metaohjelmointiin tarvittavat “kikat” ovat usein varsin vaikealukuisia ja -selkoisia, joten tällainen metaoh-

[♍] Funktiomalleissa saattaa oudoksuttaa se, että niiden kolmannesta parametrista on parametrilistassa mainittu vain tyyppi, *eikä parametrin nimeä*. Tämä on laillista C++:aa ja kertoo kääntäjälle, ettei kyseistä parametria käytetä itse koodissa lainkaan.

```
1  #include <iterator>
2  using std::iterator_traits;
3  using std::input_iterator_tag;
4  using std::bidirectional_iterator_tag;
5  using std::random_access_iterator_tag;
6
7  // Kuormitetut funktiomallit eri tapauksille
8  // Perustapaus, vähintään lukuiteraattori
9  template<typename Iteraattori, typename Siirtyma>
10 void siirry_apu(Iteraattori& iter, Siirtyma s, input_iterator_tag)
11 {
12   // Ollaan varmasti menossa eteenpäin, lukuiteraattorit eivat osaa muuta
13   for (Siirtyma i = 0; i != s; ++i)
14   {
15     ++iter;
16   }
17 }
18
19 // Erikoistapaus 1, vähintään kaksisuuntainen iteraattori
20 template<typename Iteraattori, typename Siirtyma>
21 void siirry_apu(Iteraattori& iter, Siirtyma s, bidirectional_iterator_tag)
22 {
23   // Täytyy pystyä menemään eteen- tai taaksepäin tilanteesta riippuen
24   if (s >= 0)
25   {
26     for (Siirtyma i = 0; i != s; ++i) { ++iter; }
27   }
28   else
29   {
30     for (Siirtyma i = 0; i != s; --i) { --iter; }
31   }
32 }
33
34 // Erikoistapaus 2, hajasaanti-iteraattori
35 template<typename Iteraattori, typename Siirtyma>
36 void siirry_apu(Iteraattori& iter, Siirtyma s, random_access_iterator_tag)
37 {
38   // Hypätään suoraan oikeaan paikkaan
39   iter += s;
40 }
41
42 // Varsinainen funktio, joka valitsee apufunktion metakoodilla
43 template<typename Iteraattori, typename Siirtyma>
44 inline void siirry(Iteraattori& iter, Siirtyma s)
45 {
46   siirry_apu(iter, s,
47              typename iterator_traits<Iteraattori>::iterator_category());
48 }
```

LISTAUS 10.23: Esimerkki metaohjelmoinnista optimoinnissa

jelmointi vaatii tuekseen selkeän dokumentoinnin, jotta myös koodia lukevat ymmärtävät, mitä koodin on tarkoitus tehdä. Onneksi geneerisen template-metakoodia sisältävän kirjaston *käyttäminen* on kuitenkin varsin selkeää ja usein jopa ohjelmoijalle täysin läpinäkyvää, kuten esimerkiksi STL osoittaa.

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.

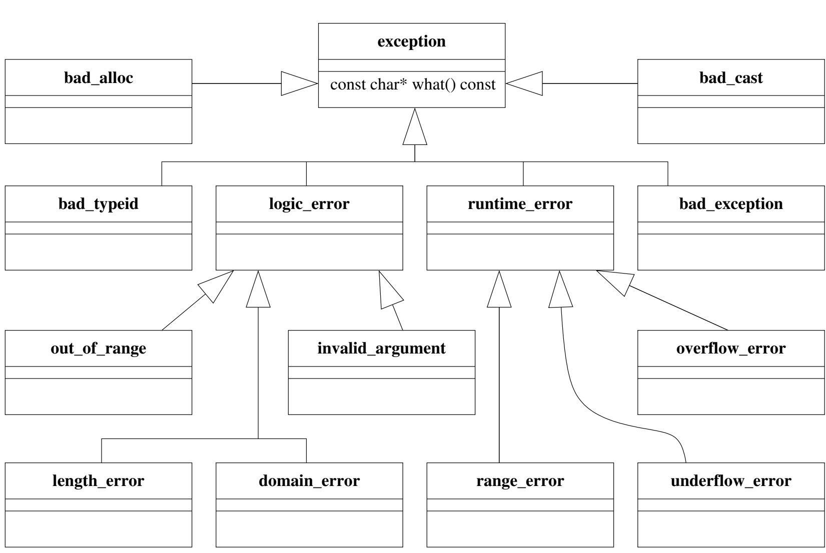
Järjestelmästä voitaisiin yrittää vapauttaa käytössä olevaa muistia, mutta resursseista järkevästi huolta pitävä ohjelmisto on tietysti alunperin toteutettu siten, ettei turhia muistivarauksia ole olemassa. Seuraavaksi voidaan etsiä “vähemmän tärkeitä” muistivarauksia ja vapautetaan ne, jolloin virhetilanne 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-

si pohjautuvat *ajoaikaisiin* tarkastuksiin siitä, millaisia poikkeuksia funktioista saa vuotaa ulos. Niinpä nykyisin monet C++-asiantuntijat suosittelevatkin, ettei poikkeusmääreitä käytettäisi, vaan funktioista mahdollisesti vuotavat poikkeukset dokumentoitaisiin muuten rajapintadokumentaatioon.

Lisää tietoa poikkeusmääreistä ja syistä niiden välttämäiseen löytyy esimerkiksi Herb Sutterin artikkeleista “A Pragmatic Look at Exception Specifications — The C++ feature that wasn’t” [Sutter, 2002a] ja “Exception Safety and Exception Specifications: Are They Worth It?” [Sutter, 2003].

Olio-ohjelmoinnissa on lisäksi useita tilanteita, joissa funktio ei tiedä, mitä kaikkia poikkeuksia voi vuotaa ulos. Tällainen tilanne voi sattua erityisesti polymorfismia ja malleja (geneerisyyttä) käytettäessä, jolloin funktio ei välttämättä tarkalleen tiedä, millaisia olioita se käsittelee, joten se ei myöskään tiedä mahdollisia poikkeuksia. On kuitenkin vaikeaa ohjelmoida niin, että varautuu kaikkiin mahdollisiin — jopa tuntemattomiin — poikkeuksiin, joten ohjelmoinnissa tulisi aina pyrkiä dokumentoimaan mahdolliset ulos vuotavat poikkeukset.

Jos funktiosta voi vuotaa ulos jotkin tunnetut poikkeukset ja lisäksi mahdollisesti muitakin, poikkeusmääreitä ei voi käyttää, koska ne edellyttävät, että poikkeuslistaan merkitään *kaikki* mahdolliset poikkeukset. Tällaisissa tapauksissa kannattaakin vain dokumentoida sopivaan ohjelmakommenttiin tunnetut poikkeukset ja lisäksi mainita, että muitakin poikkeuksia saattaa vuotaa ulos.

11.7 Muistivuotojen välttäminen: auto_ptr

Dynaamisen elinkaaren oliot tuottavat poikkeuskäsittelyssä paljon ongelmia. Niitä tarvitaan välttämättä, jos olion elinkaari ei osu yksiin minkään koodilohkon näkyvyysalueen kanssa. Toisaalta dynaamisesti luotujen olioiden tuhoaminen kaikissa mahdollisissa virhetilanteissa lisää tarvittavan koodin määrää ja tekee koodista vaikealukuisemman. Tämän vuoksi ISOC++:aan lisättiin luokka nimeltä `auto_ptr`, joka ratkaisee osan dynaamisten olioiden ongelmista. Tätä `auto_ptr`-luokkaa ei ikävä kyllä aina ole vanhahkojen kääntäjäversioiden kirjastoissa. Sen sijaan uudemmista kääntäjistä sen pitäisi löytyä.

11.7.1 Automaattiosoittimet ja muistinhallinta

Automaattiosoittimen saa käyttöönsä komennolla **#include** <memory>. Se käyttäytyy ulkoisesti lähes täsmälleen samoin kuin tavallinen osoitinkin: sen saa alustaa osoittamaan ***dynaamisesti luotuun*** olioon, siihen voi sijoittaa uuden olion, ja sen päässä olevaan olioon pääsee käsiksi normaalisti operaattoreilla * ja ->. Erona automaattiosoittimien ja tavallisten osoittimien välillä on, että automaattiosoitin ***omistaa*** päässään olevan olion. Lisäksi automaattiosoittimilla ei voi tehdä osoitinaritmetiikkaa eikä niitä voi käyttää osoittamaan taulukoihin. ***Huomaa, että automaattiosoittimen päähän saa sijoittaa vain dynaamisesti new'llä luotuja olioita!***

Omistaminen tarkoittaa sitä, että kun automaattiosoitin tuhoutuu, se suorittaa automaattisesti **delete**-operaation päässään olevalle oliolle. Automaattiosoittimien hyödyllisyys piilee juuri omistamisessa. Sen ansiosta ohjelmoijan ei tarvitse välittää dynaamisesti luotujen olioiden tuhoamisesta. Jos ne on alunperin pantu automaattiosoittimen päähän, olion tuhoamisvastuu siirtyy osoittimelle. Koska itse automaattiosoittimen elinkaari on staattinen, kääntäjä pitää huolen olion tuhoamisesta.

Listaus 11.8 seuraavalla sivulla sisältää esimerkin automaattiosoittimien käytöstä. Siitä käy ilmi automaattiosoittimien samankaltaisuus tavallisten osoittimien kanssa ja se, miten automaattiosoittimet helpottavat virhekäsittelyä verrattuna listauksen 11.7 koodiin. Koska automaattiosoitin pitää itse huolen päässään olevan olion tuhoamisesta, virhetilanteissa ei tarvitse ryhtyä mihinkään erikoistoimenpiteisiin — poikkeuksen sattuessa paikallisena muuttujana oleva automaattiosoitin tuhotaan automaattisesti, ja se puolestaan tuhoaa dynaamisesti luodun olion.

Automaattiosoittimesta saa halutessaan ulos tavallisen "ei-omistavan" osoittimen jäsenfunktiokutsulla `osoitin.get()`. Tätä voi käyttää, jos ohjelmassa tulee joskus tarve viitata olioon myös tavallisen osoittimen läpi. Tällöin on kuitenkin syytä muistaa, että olion omistus säilyy automaattiosoittimella, joten olio tuhoutuu edelleen automaattisesti automaattiosoittimen tuhoutuessa.

```
1  #include <memory>
2  using std::auto_ptr;
        ⋮
3  void siivousfunktio2()
4  {
5    auto_ptr< vector<double> > taulup(new vector<double>());
6    keskiarvo2(*taulup);
7
8    auto_ptr < vector<double> > taulu2p(new vector<double>());
9    for (unsigned int i = 0; i < taulup->size(); ++i)
10   { // Lasketaan taulukon neliöt
11     taulu2p->push_back((*taulup)[i] * (*taulup)[i]);
12   }
13   cout << "Neliöiden keskiarvo: " << laskeKeskiarvo(*taulu2p) << endl;
14 }
```

LISTAUS 11.8: Esimerkki automaattiosoittimen auto_ptr käytöstä

11.7.2 Automaattiosoittimien sijoitus ja kopiointi

Ohjelmoinnissa tulee usein vastaan tilanne, jossa monta osoitinta osoittaa samaan olioon. Automaattiosoittimien tapauksessa tämä ei kuitenkaan ole mahdollista, koska olion voi omistaa vain yksi automaattiosoitin kerrallaan. Jos automaattiosoittimia voisi osoittaa samaan olioon useita, tämä olio tulisi tuhotuksi useaan kertaan. Tämän vuoksi automaattiosoittimien kopiointi ja sijoittaminen on määritelty niin, että operaatiot ***siirtävät*** olion osoittimesta toiseen.

Siirtäminen tarkoittaa sitä, että kopioinnin jälkeen uusi osoitin osoittaa olioon ja omistaa sen kun taas vanha osoitin on "tyhjentynyt" eikä enää osoita minnekään. Vastaavasti sijoituksen yhteydessä alkuperäinen osoitin menettää olion eikä osoita enää minnekään kun taas sijoituksen kohteena oleva osoitin tuhoaa vanhan olionsa ja siirtyy omistamaan uutta oliota. Tällainen sijoituksen ja kopioinnin semantiikka, jossa myös alkuperäinen osoitin muuttuu, eroaa oleellisesti normaalista sijoituksesta ja kopioinnista, joten sen kanssa kannattaa olla tarkkana.

Omistuksen siirtyminen on kuitenkin myös erittäin hyödyllinen piirre. Sen avulla funktiossa voi luoda automaattiosoittimen päähän dynaamisesti olion ja palauttaa funktiosta `auto_ptr`-tyyppisen paluuarvon. Tällä tavoin olion omistus siirtyy paluuarvon mukana automaattisesti funktiosta kutsujan puolelle eikä kummankaan tarvitse

välittää olion tuhoamisesta edes virhetilanteissa. Listaus 11.9 näyttää, kuinka automaattiosoitinta voi käyttää hyväksi olion omistuksen siirtämiseen paikasta toiseen.

Olion omistuksen ja sen siirtymisen vuoksi automaattiosoitinta voi käyttää myös dokumentoimaan omistusta. Ohjelman voi kirjoittaa niin, että kaikki dynaamisesti luodut oliot ovat aina jonkin automaattiosoittimen päässä. Jos funktiolle täytyy välittää tieto oliosta, mutta ei omistusvastuuta, käytetään joko viitettä tai tavallista osoitinta. Jos taas funktiolle halutaan siirtää myös omistusvastuu, välitetään automaattiosoitin. Tällaisessa ohjelmoinnissa dynaamisesti luotuja olioita ei tarvitse koskaan tuhota ja ainakin periaatteessa muistivuodot ovat mahdottomia. Käytännössä ohjelmoijan täytyy kuitenkin varmistua siitä, etteivät tavalliset osoittimet tai viitteet jää viittaamaan jo tuhottuun olioon ja ettei jo siirrettyyn olioon yritetä päästä

```
12 typedef auto_ptr< vector<double> > AutoTauluPtr;
13
14 AutoTauluPtr tuotaTaulukko()
15 {
16   AutoTauluPtr taulup(new vector<double>());
17   lueLuvutTaulukkoon(*taulup);
18   return taulup;
19 }
20 AutoTauluPtr tuotaNeliotaulukko(vector<double> const* taulup)
21 { // Parempi vaihtoehto: tuotaNeliotaulukko(vector<double> const& taulu)
22   AutoTauluPtr neliop(new vector<double>());
23   for (unsigned int i = 0; i < taulup->size(); ++i)
24   { // Lasketaan taulukon neliöt
25     neliop->push_back((*taulup)[i] * (*taulup)[i]);
26   }
27   return neliop;
28 }
29
30 void siivousfunktio3()
31 {
32   AutoTauluPtr taulu1p(tuotaTaulukko()); // Taulukon omistus siirtyy tänne
33   AutoTauluPtr taulu2p(tuotaNeliotaulukko(taulu1p.get())); // Samoin tässä
34
35   cout << "Keskiarvo: " << laskeKeskiarvo(*taulu1p) << endl;
36   cout << "Neliöiden keskiarvo: " << laskeKeskiarvo(*taulu2p) << endl;
37 }
```

LISTAUS 11.9: Automaattiosoitin ja omistuksen siirto

käsiksi vanhan automaattiosoittimen kautta.

11.7.3 Automaattiosoittimien käytön rajoitukset

Aiemmin tässä luvussa on jo tullut esille joitain automaattiosoittimia koskevia rajoituksia. Seuraavaan luetteloon on kerätty tärkeimmät rajoitukset.

- Automaattiosoittimen päähän saa laittaa vain dynaamisesti **`new`**'llä luotuja olioita.
- Vain yksi automaattiosoitin voi osoittaa samaan olioon kerrallaan. Automaattiosoittimien sijoitus ja kopiointi siirtävät olion omistuksen automaattiosoittimelta toiselle.
- Sijoituksen ja kopioinnin jälkeen olioon ei pääse käsiksi vanhan automaattiosoittimen kautta.
- Automaattiosoittimelle ei voi tehdä osoitinaritmetiikkaa (`++`, `--`, indeksointi ynnä muut).
- Automaattiosoittimen päähän ei voi panna tavallisia taulukoita (sen sijaan `vector` ei tuota ongelmia).
- Automaattiosoittimia ei voi laittaa STL:n tietorakenteiden sisälle. Tämä tarkoittaa, että esimerkiksi `vector< auto_ptr<int> >` ei toimi.

Nämä rajoitukset huomioon ottamalla automaattiosoittimet helpottavat huomattavasti dynaamisten olioiden hallintaa. Rajoituksista ehkä ikävin on se, ettei automaattiosoittimia voi panna suoraan STL:n tietorakenteisiin. Syy tähän on se, että STL vaatii että tietorakenteiden alkioita voi kopioida ja sijoittaa normaalisti. Automaattiosoittimien omistuksen siirto aiheuttaa sen, että sijoituksessa ja kopioinnissa alkuperäinen olio muuttuu, joten automaattiosoittimet eivät ole yhteensopivia STL:n kanssa.

Vaihtoehtoja automaattiosoittimille ovat "**älykkäät osoittimet**" (*smart pointer*), jotka käyttävät viitelaskureita olion elinkaaren määräämiseen. Niiden idea on, että älykkäät osoittimet pitävät yllä laskuria siitä, kuinka monta osoitinta olioon osoittaa. Siinä vaiheessa, kun viimeinen älykäs osoitin tuhoutuu tai lakkaa muuten osoittamasta olioon, se tuhoaa olion. Älykkäät osoittimet eivät kuulu ISOC++ -kieleen, mutta verkosta saa useita toimivia toteutuksia.

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 kaikkia 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.

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

```
1 Kirja& Kirja::operator =(Kirja const& kirja)
2 {
3   if (this != &kirja)
4   {
5     nimi_ = kirja.nimi_;
6     tekija_ = kirja.tekija_;
7   }
8   return *this;
9 }
```

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)[♊] ja se on joskus varsin käyttökelpoinen muutenkin kuin poikkeuksien yhteydessä [Sutter, 2000, kohdat 26–30].

[♊] 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

Liite A

C++: Ei-olio-ominaisuuksia

Vaikka sana “ei-olio-ominaisuuksia” onkin melkoinen hirvitys, se kuvaa hyvin sitä, miksi nämä C++-ohjelmointikielen rakenteet on esitelty lyhyesti erillisessä liitteessä. Kyseessä on joukko kielen ominaisuuksia (lähinnä parannuksia C-kieleen), jotka C++:n käyttäjän on hyvä tuntea, mutta joilla ei ole *suoraan* mitään tekemistä kirjan aiheen eli olio-ohjelmoinnin kanssa.

Esiteltävistä asioista viitteet ovat uusi “osoitustyyppi”, **inline** on puhtaasti optimointiominaisuus ja vector sekä string ovat uusia helppokäyttöisempiä versioita C:ssä olevista ominaisuuksista.

Tämän liitteen tarkoituksena on antaa näistä C++:n ominaisuuksista sen verran tietoa, että muun kirjan lukeminen ja ymmärtäminen onnistuvat ilman suuria ongelmia. Kattavan kuvauksen näistä ominaisuuksista saa parhaiten jostain C++-oppikirjasta [Lippman ja Lajoie, 1997], [Stroustrup, 1997].

A.1 Viitteet

C-kieli käyttää kahdenlaisia tapoja tiedon käsittelyyn ja välittämiseen esimerkiksi funktioiden parametreina: muuttujat ja osoittimet. Muuttujat ovat varsinainen datan esitysmuoto, ja osoittimien avulla voidaan viitata olemassa olevaan dataan (muuttuja). Osoittimet ovat toteutukseltaan muistiosoitteita, ja niitä pystyy käsittelemään hyvin vapaasti: vaihtamaan arvoa mielivaltaiseksi kokonaisluvuksi ja siirtä-

mään osoittamaan esimerkiksi "seuraavaan" alkioon (++-operaattori). Näistä vapauksista johtuen osoittimet ovat myös usein pahojen ohjelmointivirheiden lähde. Ohjelmointivirheen seurauksena väärään paikkaan osoittava osoitin voi aiheuttaa muutoksia ohjelman datassa väärässä paikassa tai jossain vaiheessa järkevä osoitinarvo osoittaakin muistin vapauttamisen jälkeen kielletylle alueelle, jota silti yritetään käyttää osoittimen läpi. Nämä virheet paljastuvat usein vasta hyvin pitkien aikojen kuluttua (ohjelman aivan muut kuin virheen aiheuttaneet osat alkavat käyttäytyä virheellisesti).

Osoittimien "vaarallisuuden" takia C++ määrittelee uuden tietotyypin **viite** (*reference*), joka osoittimien tapaan viittaa olemassa olevaan dataan mutta johon liittyen kääntäjä tekee enemmän käyttötarkistuksia kuin osoittimien yhteydessä. Viite on tavallaan synonyymi olemassa olevalle data-alkiolle, ja jokainen viite viittaa ***aina*** johonkin dataan (se ei voi olla nolla kuten osoitin).[ϒ]

Aina kun viite luodaan, se täytyy samalla alustaa osoittamaan aikaisemmin luotuun dataan. C++ uudelleenkäyttää viitteiden yhteydessä hieman hämäävästi osoittimiin liittyvän syntaktisen merkin "&". Kun muuttujan *tietotyypin perään* on lisätty &, kyseessä on viitetyyppi kyseistä tyyppiä olevaan alkioon:

```
int a = 0;
int b = 0;          // normaali kokonaislukumuuttujien esittely
int& a_r = a;       // kokonaislukuviite muuttujaan a
```

Kun viitetyyppistä muuttujaa käytetään, se käyttytyy samoin kuin kohdassa esiintyisi alkuperäinen muuttuja (viite on toinen muoto eli synonyymi alkuperäiselle muuttujalle).

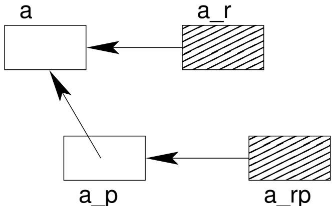
```
b = a_r + 1;      // lukee muuttujan a arvon ⇒ b == 1
a_r = 2 - b;      // kirjoittaa muuttujaan a ⇒ a == 1
```

Osoittimeen voi tehdä viitteen, mutta tyyppiä "osoitin viitteeseen" ei ole olemassa(!) (katso kuva A.1 seuraavalla sivulla).

```
int*  a_p  = &a_r; // osoittaa muuttujaan a (ei viitteeseen!)
int*& a_rp = a_p;  // synonyymi osoittimelle

*a_rp = 7;  // muuttujan a arvo on sijoituksen jälkeen 7
```

[ϒ] Tässä on suuri ero Javan viitetyyppeihin, joilla on olemassa tyhjä viitearvo **null**.



KUVA A.1: Esimerkkien viittaukset

Viitteiden normaali käyttötarkoitus ei ole luoda turhia uusia nimisynonyymejä jo olemassa oleville muuttujille vaan välittää viittauksia dataan funktioiden parametreina tai paluuarvoina. Viiteparametri sitoutuu synonyymiksi funktiolle annettuun dataan, kun funktiota kutsutaan:

```
void tuplaa( int& i ) { i = i * 2; }
     ⋮
int x = 2;
tuplaa( x );
// x on nyt 4
```

Viiteparametrin avulla on helpompi kirjoittaa funktioita, jotka muuttavat parametriensa arvoja. Haittapuoli taas on se, että funktion kutsusta `tuplaa( x )` ei pysty päättelemään mitenkään, muuttaako funktio parametrina annettua arvoa. Tämä tieto on tarkistettava funktion esittelystä, jossa funktio voi luvata vakioviiteparametrilla, että se ei muuta parametrin arvoa:

```
struct SuuriData { ... };

void arvonValitys( SuuriData d );
void vakioViiteValitys( SuuriData const& d );
```

Kumpikaan esitellyistä funktioista ei muuta parametrina annettua dataa. `ArvonValitys` kopioi tietorakenteen datasta itselleen oman kopion, jolloin kutsujan antama data ei muutu. `VakioViiteValitys` ei myöskään muuta dataa, koska se on merkitty kääntäjälle vakioda-

taksi. Mahdollisesti raskasta datan kopiointioperaatiota ei kuitenkaan tehdä, vaan funktiolle välittyy käyttöön viite alkuperäiseen dataan.

Funktio voi palauttaa viitteen olemassa olevaan dataan myös paluuarvonaan. Tämän ominaisuuden avulla voidaan tehdä funktio, joka valitsee palautettavan arvon, joka on jatkokäsiteltävissä aivan kuin käytössä olisi koko ajan ollut alkuperäinen arvo:

```
1 int  data[10];
2 int& kohdistin( int i ) { return data[i]; }
        ⋮
3 kohdistin(0) = 7; // sijoittaa taulukon ensimmäiseen alkioon
```

Funktio `kohdistin` palauttaa viitteen, joka on sitoutunut siihen taulukon `data` alkioon, jonka funktion parametri `i` määrää. Koska paluuarvo käyttäytyy kuten alkuperäinen alkio, siihen voidaan tehdä myös rivin 3 mukainen sijoitus.

Viitepaluuarvoihin liittyy myös tilanne, jossa saadaan aikaiseksi virheellinen viite. Koska funktion paikallinen data tuhoutuu funktion suorituksen päätyttyä, tällaiseen dataan palautettava viite ei viittaa olemassa olevaan dataan, ja viitteen käyttö aiheuttaa määrittelemättömän toiminnon:

```
int& virheellinen( int a )
{
  int b = 2 * a;
  return b; // VIRHE! paikallinen muuttuja viitepaluuarvona
}
```

A.2 inline

Useimmissa prosessoriarkkitehtuureissa funktiokutsu on eniten käytetyistä operaatioista raskain (parametrien sijoitus esimerkiksi pinoon ja arvokopioinnit funktion alussa ja lopussa). C++-ohjelmoija voi merkitä haluamansa funktiot avainsanalla **`inline`**, joka antaa kääntäjälle luvan optimoida funktion käyttöä.

Optimoinnissa kääntäjä yrittää korvata funktion kutsun sen sisältämällä toiminnallisuudella. ***Tämä tapahtuu siten, että funktion alkuperäinen toiminnallisuus ei saa muuttua mitenkään.*** Jos funktio

on liian mutkikas tällaiseen korvaukseen, kääntäjä suorittaa normaalin funktiokutsun. Ohjelmoijan ei tarvitse välittää siitä, mitä mahdollisia optimointimuutoksia **inline**n käyttö aiheuttaa syntyvään konekoodiin — toiminnallisuus on edelleen sama, kuin jos mitään optimointia ei olisi tehty. C++-standardi ei vaadi kääntäjiltä optimointien suorittamista, joten C++-kääntäjä, joka jättää **inline**-avainsanat ottamatta huomioon, on standardin mukaisesti toimiva. (Tietysti juuri tällaiset kohdat standardissa antavat kääntäjävalmistajille kilpailuvaraa, joten käytännössä kaikki C++-kääntäjät suorittavat **inline**-optimointeja.)

Esimerkiksi ohjelmakoodissa

```
1 inline int max( int x, int y )
2 {
3   if( x > y ) { return x; }
4   else { return y; }
5 }
        ⋮
6   return max( a, max( b, c ) );
```

rivi 6 voi **inline**-funktiokutsujen takia muuttua muotoon:

```
{ int r1_=(b>c)?b:c;
  int r2_=(a>r1_)?a:r1_;
  return r2_;}
```

Käytännössä optimoinnit suoritetaan kääntäjän tuottaman konekoodin tasolla, mutta esimerkistä näkyy **inline**n idea funktiokutsujen poistamisessa.

A.3 vector

Taulukko on tietorakenne, joka sisältää sarjan samaa tietotyyppiä olevia alkioita. Taulukoiden perusominaisuus on indeksointi, jossa voidaan viitata (yhtä nopeasti) taulukon mihin tahansa alkioon (`T[i]`). [Sethi, 1996, luku 4.4]

C-kielessä taulukot ja osoittimet ovat "lähisukulaisia". Taulukon nimi toimii osoittimena taulukon ensimmäiseen alkioon. Erityisesti

tämä piirre korostuu funktioiden parametreissa, joissa välitetään taulukoita osoittimina, vaikka syntaksi saattaisikin viitata taulukon välittämiseen arvona: [Kerninghan ja Ritchie, 1988, luku 5.3]

```
void poke( char s[] ) { s[1] = 'u'; }
     ⋮
char j[] = "Jyke";
poke( j );
// j on nyt "Juke";
```

C++ tarjoaa C-kielen tyyppisten taulukoiden lisäksi uuden taulukkotyypin "vektori", joka ei ole taulukoiden tavoin sitoutunut osoittimiin.

```
#include <vector> // taulukkotyypin esittely käyttöön

int main()
{
  int vt[10];            // "vanha" taulukko kokonaislukuja
  vector<int> ut(10); // vektoritaulukko kokonaislukuja
}
```

C-taulukoihin verrattuna vektori on turvallista välittää arvoparametrina (se kopioituu), ja vektoriin voidaan sijoittaa toinen vektori. Vektorin koko kasvaa tarvittaessa (automaattinen tilan kasvatus), mikä vähentää muistinhallinan ongelmia. Koska vektorin tilanvarausta ei ole C-taulukoiden tapaan pakko ilmoittaa luonnin yhteydessä, se voidaan luoda tyhjänä ja täyttää tarvittavilla alkioilla. Taulukon rajapinnassa oleva operaatio push_back lisää uuden alkion taulukon loppuun:

```
vector<unsigned int> taulukko; // tyhjä vektori

for( unsigned int i=1; i<=10; ++i ) // lisätään alkiot 1..10
  { taulukko.push_back(i); }
```

Vektoritaulukoita indeksoidaan samalla syntaksilla kuin C-taulukoitakin (ut[i]), ja samoin kuin aikaisemmin, tässä indeksointitavassa ei ole tarkastuksia indeksoinnin "onnistumisesta" eli osumisesta taulukon alueelle. Sekä vt[11] että ut[11] ovat C++-standardin

kannalta määrittelemättömiä operaatioita (viittaus tietorakenteen ulkopuolelle). Vektori tarjoaa rajapinnassaan operaation `at`, joka suorittaa ajoaikaisen indeksoinnin oikeellisuuden tarkastuksen. Lause `ut.at(11)` tuottaa siis virheen (poikkeuksen avulla, katso luku 11).

Kuvassa A.2 on vertailtu C-taulukon ja vektorin toimintoja.

A.4 string

C-kielen yhteydessä merkkijonoista puhuttaessa tarkoitetaan merkkitaulukkoa (**char**[] tai **unsigned char**[]), jonka viimeisenä (päättymismerkkinä) on sovittu olevan merkkikoodi nolla ('\0'). C++ tarjoaa uuden merkkijonotietotyypin `string`, joka on "täysiverinen" tietotyyppi eikä enää kielen alimmalla tasolla oleva yhdistetty taulukko- ja osoitin-rakenne C:n tapaan.

`string`-tyyppiset muuttujat muun muassa kopioituvat funktioi-

Toiminta	**C-taulukko**	**vektori**
Luominen	**int** t[10]	vector<**int**> t(10)
Luominen, tyhjä	(ei järkevä)	vector<**int**> t
Sijoitus	(silmukassa)	t1 = t2
Indeksointi	t[i]	t[i] tai t.at(i)
Lisäys loppuun	(mahdoton)	t.push_back(a)
Koko	(tiedettävä muualta)	t.size()
Tyhjennys	(mahdoton)	t.clear()
Vertailu	(silmukassa)	t1 == t2 (myös t1 < t2, yms.)
Lisäys keskelle	(mahdoton)	t.insert(paikka-iteraattori, a)
Etsiminen	(silmukassa)	find(t.begin(), t.end(), a)
Arvojen vaihto	(silmukassa)	t1.swap(t2)

KUVA A.2: C-taulukon ja vektorin vertailu

Toiminta	merkkijonotaulukko	string
Alustus	`char* s = "jaa";`	`string s("mää");`
Sijoitus	`strcpy(mihin, mista)`	`mihin = mista`
Indeksointi	`s[i]`	`s[i]` tai `s.at(i)`
Yhdistäminen	`strcat(mihin, mika)`	`mihin += mika`
Alimerkkijono	`strncpy(t,&s[alku],koko)`	`t=s.substr(alku,alku+koko)`
Koko	`strlen(s)`	`s.length()`
Tyhjennys	`s[0] = '\0'`	`s.clear()`
Vertailu	`strcmp(s, t)`	`s == t` (myös `s < t` yms.)
Rivin luku	(eri tapoja)	`getline(cin, s)`
Lisääminen	(todella hankalaa)	`s.insert(paikka-iter, t)`
Etsiminen	`strstr(mista, mita)`	`mista.find(mita)`
Arvojen vaihto	`char* tt=s; s=t; t=tt;`	`s.swap(t)`

KUVA A.3: Merkkijonotaulukon ja C++:n `string`in vertailu

```
 1 #include <cstdlib>
 2 #include <string>
 3 using std::string;
 4 #include <iostream>
 5 using std::cin;
 6 using std::cout;
 7 using std::endl;
 8
 9 int main()
10 {
11     string sana, kokoelma("ALKU");
12
13     while( cin >> sana ) {
14         if( sana.find("ja") != string::npos ) {
15             // löytyi
16             kokoelma = kokoelma + "," + sana;
17         }
18     }
19     cout << kokoelma << endl;
20   return EXIT_SUCCESS;
21 }
```

LISTAUS A.1: Osamerkkijonon "ja" etsintä (C++)

```
 1 #include <string.h>
 2 #include <stdlib.h>
 3 #include <stdio.h>
 4
 5 int main()
 6 {
 7     char sana[1024]; /* ei toimi tätä pidemmillä sanoilla */
 8     char *kokoelma, *tmp;
 9     unsigned int pituus, kaytossa;
10     pituus = kaytossa = 5; /* ALKU + lopetusmerkki */
11     kokoelma = (char*)malloc( pituus ); strcpy( kokoelma, "ALKU" );
12     while( scanf("%s", sana) == 1) {
13         if( strstr(sana, "ja") != 0 ) {
14             if( kaytossa + strlen(sana) + 2 > pituus ) {
15                 /* muisti ei riitä -> lisää */
16                 tmp = kokoelma;
17                 kokoelma = (char*)malloc( pituus + 2*strlen(sana) );
18                 strcpy( kokoelma, tmp );
19                 free( tmp );
20                 pituus += 2*strlen(sana);
21             }
22             strcat( kokoelma, "," );
23             strcat( kokoelma, sana );
24             kaytossa = kaytossa + strlen(sana) + 2;
25         }
26     }
27     printf("%s\n", kokoelma);
28     return EXIT_SUCCESS;
29 }
```

LISTAUS A.2: Osamerkkijonon “ja” etsintä (C)

den parametreina:

```
#include <string> // merkkijonotyypin esittely käyttöön

void spoke( string s ) { s[1] = 'u'; }
      ⋮
string j = "Jyke";
spoke( j );
// j:n arvo on edelleen “Jyke”
```

C-kielen kirjasto määrittelee merkkijonotaulukoiden käsittelyyn joukon str-alkuisia funktioita, joiden yhteydessä ohjelmoijan on jat-

kuvasti itse huolehdittava siitä, että operaatioiden kohdemerkkijonoissa on tarpeeksi tilaa (merkkitaulukko on tarpeeksi suuri operaation tuloksen säilyttämiseen). C++:n stringin rajapinnassa ovat kaikki vastaavat operaatiot (vertailu kuvassa A.3 sivulla 417), ja niiden yhteydessä mahdolliset tulosmerkkijonon koon muutokset tapahtuvat automaattisesti.

Listauksessa A.1 sivulla 417 on esimerkki string-tietorakenteen käytöstä. Ohjelma lukee oletussyötteestään sanoja ja muodostaa pilkulla erotellun luettelon (merkkijonoon) kaikista niistä sanoista, jotka sisältävät sanan “ja”. Lopuksi ohjelma tulostaa muodostamansa luettelon. Listauksessa A.2 edellisellä sivulla sama toiminnallisuus on toteutettu C-kielellä merkkijonotaulukoiden avulla, jolloin ohjelmoijan on itse huolehdittava muistinhallinnasta: etukäteen ei ole tiedossa syötteen yksittäisen sanan tai syntyvän luettelon pituutta. Käsiteltävän sanan maksimipituuden ongelma on ratkaistu “taikavakiolla” 1024, ja tuotettavan listan muistinhallinta on ohjelmoitu itse malloc- ja free-rutiineilla.

Liite B

C++-tyyliopas

Opettele säännöt, jotta tiedät miten rikkoa niitä oikein.
– Tuntematon

Koska nykyaikaiset ohjelmointikielet ovat mutkikkaita ja monitahoisia työkaluja, tyylioppaalla pyritään rajaamaan projektin tai organisaation tarpeisiin sopivat tavat käyttää tätä työkalua. Yhtenäisen ohjelmointityylin noudattaminen lisää lähdekoodin ymmärrettävyyttä, ylläpidettävyyttä, siirrettävyyttä, luettavuutta ja uudelleenkäytettävyyttä. Mutkikkaissa ohjelmointikielissä (kuten C++) tyylioppaan ohjeet voivat myös lisätä koodin tehokkuutta ja auttaa välttämään ohjelmointivirheitä — tai ainakin auttaa löytämään virheet nopeammin. Hyvä tyyliopas pystyy parantamaan kirjoittamamme ohjelmakoodin laatua. Tähän tyylioppaaseen on koottu sääntöjä, joita noudattamalla pääsee kohti tuota päämäärää käytettäessä C++-ohjelmointikieltä.

Opas on kokoelma sääntöjä ja suosituksia. Säännöt ovat perusteltavissa ISOC++ -standardilla tai muilla erityisen painavilla syillä. Ohjelmakoodin ulkoasuun, muokkaamiseen ja taltiointiin liittyvät suositukset on tarkoitettu ***yhdenmukaisiksi sopimuksiksi***, joita tarvittaessa muokataan tyyliohjetta sovellettaessa. Suositus "tiedostonimi päättyy aina päätteeseen `.cc`" voidaan tarvittaessa muokata tarkoittamaan esimerkiksi tiedostopäätettä `.C`, `.cxx` tai `.cpp` käytetyn käännösympäristön vaatimusten mukaisesti.

1. **Yleistä**

1.1. Kaikkia tyylioppaan sääntöjä on noudatettava.

1.2. Tyylioppaan säännöstä saa aina poiketa hyvällä perusteella, joka tulee kirjata ohjelmakoodin kommentteihin.

1.3. Säännöt eivät koske ulkopuolista lähdekoodia (muualta hankittu tai koodigeneraattorin tuottama).

1.4. Vanhoja ohjelmia ei muuteta tämän tyylioppaan mukaisiksi, vaan niitä ylläpidetään niitä tehtäessä noudatetun käytännön mukaisesti.

2. **Ohjelmiston tiedostot**

2.1. Tiedostot ovat määrämuotoisia: alussa on koko tiedostoa koskeva kommentti.

2.2. Tiedoston kommentit kirjoitetaan C++-muodossa (`// ...`). Monen rivin kommentit voidaan sijoittaa C-kielen kommenttien (`/* ... */`) sisään.

2.3. Otsikkotiedostot

2.3.1. Otsikkotiedostojen nimen pääte on `.hh`

Yhtenäiseen nimeämiseen kuuluvat määrämuotoiset tiedostonimet. Päätteen valintaan vaikuttaa ensisijaisesti se, mitä tiedostoja käytetty käännösympäristö pitää oletusarvoisesti C++-tiedostoina. Päätettä `.h` ei kannata käyttää, koska se on jo käytössä C-kielisissä ohjelmissa.

2.3.2. Yhdessä otsikkotiedostossa esitellään yksi julkinen rajapinta (luokka tai nimiavaruus).

Hyvin dokumentoitu julkinen rajapinta on usein sopivan kokoinen kokonaisuus sijoitettavaksi yhteen ohjelmakooditiedostoon. (Katso myös kohta 5.1.)

2.3.3. Otsikkotiedosto on selkeästi kommentoitu, jos sen perusteella pystytään suunnittelemaan ja toteuttamaan rajapintaa koskevat testitapaukset.

2.3.4. Otsikkotiedostoissa on ainoastaan esittelyitä. (ISOC++:n ominaisuuksien takia otsikkotiedostossa voi olla myös kokonaislukuvakioita, **`inline`**-rakenteita [aliluku A.2 sivulla 413] ja **`template`**-malleja [aliluku 9.5 sivulla 283].)

*Vaikka **`inline`**- ja **`template`**-rakenteet sisältävät toteutuksen ohjelmakoodia, niin nykyisten C++-kääntäjien on nähtävä niiden täysi määrittely, ennen kuin kyseisiä rakenteita voidaan käyttää (instantioida) muussa ohjelmakoodissa.*

2.3.5. Laajat mallit ja **inline**-koodit tulee kirjoittaa erilliseen tiedostoon (jolla on yhtenäinen pääte esimerkiksi `.icc`). Tämä tiedosto luetaan otsikkotiedoston lopussa `#include`-käskyllä.

Toteutustapa jolla sääntöä 2.3.4 voidaan noudattaa selkeästi.

2.3.6. Otsikkotiedostossa otetaan käyttöön (**#include**-käskyillä) ainoastaan ne muut esittelyt, jotka ovat tarpeen tiedoston sisältämän esittelyn takia.

Ylimääräiset `#include`-käskyt aiheuttavat turhia riippuvuuksia lähdekooditiedostojen välille vaikeuttaen ylläpidettävyyttä.

2.3.7. Otsikkotiedostoissa ei saa käyttää **using**-lauseita. [aliluku 1.5.7 sivulla 47] (Katso myös kohta 3.3.)

Esittelyiden yhteydessä on suurin vaara nimikonflikteista eri otsikkotiedostojen sisältämien nimien välillä. Eksplisiittinen nimien käyttö näkyvyystarkentimella ei sotke ohjelman nimiavaruuksia tai tuota ohjelmakoodia, jonka toimivuus riippuu käytettyjen otsikkotiedostojen keskinäisestä käyttöönottojärjestyksestä.

2.3.8. Jos esittelyssä (yleensä otsikkotiedostossa) viitataan johonkin luokkatyyppiin vain osoittimella tai viitteellä, käytetään ennakkoesittelyä [aliluku 4.4 sivulla 112]. ***Ennakkoesittelyn saa tehdä vain itse tekemilleen luokille***.

*Vain itse tehdystä ohjelmakoodista voi varmasti tietää ennakkoesittelyn olevan sallittua. Esimerkiksi vaikka standardikirjaston `std::string` [aliluku A.4 sivulla 416] vaikuttaa luokkatyypiltä, siihen ei saa tehdä ennakkoesittelyä, koska toteutus on tyyppialias (**typedef**).*

2.3.9. Otsikkotiedosto on suojattava moninkertaiselta käyttöönotolta [aliluku 1.4 sivulla 39].

`#include`-ketjuissa moninkertainen otsikkotiedoston käyttöönotto on mahdollista ja aiheuttaa käännösvirheen (C++-rakenteet voidaan esitellä kääntäjälle vain kerran saman käännöksen yhteydessä). Suojaus toteutetaan ehdollisen kääntämisen esiprosessorikäskyillä otsikkotiedoston alussa ja lopussa:

```
#ifndef LUOKKA_A_HH
#define LUOKKA_A_HH
        ⋮
#endif // LUOKKA_A_HH
```

2.3.10. `#include`-käskyissä ei saa esiintyä viittauksia tietyn koneen tiettyyn hakemistoon.

Kaikki yhteen käännösympäristöön sitovat viittaukset vaikeuttavat lähdekoodin ylläpitoa ja siirrettävyyttä.

2.4. Toteutustiedostot

2.4.1. Toteutukset sisältävän tiedoston pääte on `.cc`

2.4.2. Toteutustiedostossa otetaan käyttöön vain kyseisen käännösyksikön toteuttamisen kannalta tarpeelliset otsikkotiedostot.

2.4.3. Toteutustiedostossa otetaan ensin käyttöön ohjelman paikalliset esittelyt (otsikkotiedostot) ja vasta sitten käännösympäristön ja ISOC++-kielen määrittelemät kirjastot [aliluku 1.5.7 sivulla 47] (katso myös kohta 3.3.5).

Ulkopuolisten esittelyiden keskinäisestä järjestyksestä riippuvat otsikkotiedostot ovat vaarallista ohjelmakoodia ja tällä käytännöllä ne huomataan helpommin käännöksen yhteydessä.

2.4.4. Rajapinnan toteutukset esitetään tiedostossa samassa järjestyksessä kuin ne ovat vastaavassa otsikkotiedostossa.

Näin määrätyn rajapintaoperaation toteutuksen löytäminen helpottuu.

3. Nimeäminen

3.1. Ohjelmakoodin kommentit ja symboliset nimet kirjoitetaan yhtenäisellä kielellä ja nimeämistavalla.

Esimerkiksi TTY:n opetuskieli on suomi, joten on loogista käyttää samaa kieltä myös ohjelmoidessa. Vastaavasti jos yrityksen virallinen kieli on englanti, se lienee parempi vaihtoehto ohjelmakoodin nimien ja kommentoinnin kieleksi.

3.2. Alaviivalla alkavia nimiä ei saa käyttää. Samoin kiellettyjä ovat nimet, joissa on kaksi peräkkäistä alaviivaa. [aliluku 2.3.2 sivulla 61].

3.3. Nimiavaruudet

3.3.1. Moduulin käyttämät nimet pidetään erillään muusta ohjelmistosta sijoittamalla ne nimiavaruuden (**`namespace`**) sisään [aliluku 1.5.1 sivulla 42].

3.3.2. ISOC++:n uusia otsikkotiedostoja ja niiden kautta std-nimiavaruutta on käytettävä. [aliluku 1.5.4 sivulla 45] ja [aliluku 1.5.5 sivulla 45].

3.3.3. Jos nimiavaruuden sisältä nostetaan näkyville nimiä, niin käyttöön otetaan vain todella tarvittavat nimet (lauseen **using namespace** X käyttö on kiellettyä). [aliluku 1.5.7 sivulla 47]

3.3.4. **using**-lauseella nostetaan nimet käyttöön lähelle käyttöpaikkaa (koodilohko tai funktio).

3.3.5. Erityisen usein tarvittavat nimet voidaan nostaa näkyville koko käännösyksikköön ja niitä koskevat **using**-lauseet tulee sijoittaa tiedoston kaikkien #include-käskyjen jälkeen, jotta “nostetut” nimet eivät pääse sotkemaan toisia esittelyitä. ISOC++:n otsikkotiedostoja voidaan kuitenkin pitää hyvin muotoiltuina ja niiden yhteyteen liitetyt kyseistä esittelyä koskevat **using**-lauseet usein parantavat ohjelmakoodin luettavuutta. [aliluku 1.5.7 sivulla 47]

```
#include "prjlib.hh"
#include <iostream>
using std::cout;
using std::endl;
#include <sstream>
using std::ostringstream;

using Prjlib::Puskuri;
using Prjlib::Paivays;
```

3.3.6. Käännösyksikön (tiedosto) sisäisessä käytössä olevat määrittelyt sijoitetaan nimeämättömän nimiavaruuden sisään. [aliluku 1.5.8 sivulla 49]

Oletuksena käännösyksikön nimet voivat aiheuttaa nimikonflikteja muun ohjelmiston osien kanssa linkitysvaiheessa. Nimeämätön nimiavaruus varmistaa määrittelyjen olevan uniikkeja (käännösyksikön sisäisiä).

```
namespace { // nimeämätön nimiavaruus
  unsigned long int viiteLaskuri = 0;
  void lisaaViiteLaskuria() {
       ⋮
  }
}
```

3.4. Luokka- (**class**) ja tietuetyyppien (**struct**), nimiavaruuksien (**namespace**) sekä tyyppialiaksien (**typedef**) nimet alkavat isolla alkukirjaimella.

3.5. Muuttujien, funktioiden ja jäsenfunktioiden nimet alkavat pienellä alkukirjaimella. Useammasta sanasta koostuvat nimet kirjoitetaan yhteen ja loppupään sanat aloitetaan isolla alkukirjaimella (esimerkiksi: `haeRaportinPaivays()`). ***Poikkeus sääntöön on luokan rakentaja, jonka nimen on oltava täsmälleen sama kuin luokan nimi***.

3.6. Jäsenmuuttujien nimen viimeinen merkki on alaviiva (tai ne erotellaan muista nimistä jollain muulla yhtenäisellä nimeämistavalla) [aliluku 2.3.2 sivulla 61].

3.7. Luokkamuuttujien nimen viimeinen merkki on alaviiva (tai ne erotellaan muista nimistä ja mahdollisesti myös jäsenmuuttujista jollain muulla yhtenäisellä tavalla) [aliluku 8.2.2 sivulla 250].

Kohdan 3.6 mukainen nimeäminen, joka helpottaa luokkamuuttujien tunnistamista ohjelmakoodissa.

3.8. Vakiot kirjoitetaan kokonaan isoilla kirjaimilla ja sanojen erottimena käytetään tarvittaessa alaviivaa. ("Vakion" arvo on aina sama. Jos esim. funktion paikallisen const-muuttujan arvo lasketaan ajoaikana ja voi vaihdella funktion kutsukerroilla, kannattaa se varmaan nimetä tavallisen muuttujan tapaan.)

```
unsigned int const PUSKURIN_SUURIN_KOKO = 1024;
```

4. Muuttujat ja vakiot

4.1. Muuttujien näkyvyysalue tulee suunnitella mahdollisimman pieneksi (koodilohko, funktio, luokka tai nimiavaruus).

Lähellä käyttöpaikkaa määritellyt ja alustetut muuttujat lisäävät ohjelmakoodin selkeyttä.

4.2. Jokainen muuttuja on määriteltävä eri lauseessa.

Muuttujan tyyppi, nimi, alustus ja sitä koskeva kommentti ovat usein yhdelle riville sopiva kokonaisuus. Sääntö myös varmistaa, että kohdan 5.4 mukaiset osoittimien ja viitteiden määrittelyt tulevat aina oikein. "`char* p1, p2;`" olisi luultavasti virhe, sillä nyt `p1` on osoitin ja `p2` merkkimuuttuja.

4.3. Ohjelmassa ei saa olla alustamattomia muuttujia.

Jos muuttujalla ei ole järkevää alkuarvoa, sille annetaan jokin "laiton" alkuarvo (nolla, miinus yksi, tms.). Alustamattomat muuttujat ovat yksi suurimmista (satunnaisten) virhetoimintojen aiheuttajista ohjelmistoissa. Laittomaan arvoon alustetut muuttujat eivät korjaa virheitä, mutta tuovat ne testeissä helpommin esille.

4.4. Luokan jäsenmuuttujat alustetaan aina rakentajan alustuslistassa, jossa jäsenmuuttujat luetellaan niiden esittelyjärjestyksessä [aliluku 3.4.1 sivulla 80].

4.5. **#define**-käskyä ei saa käyttää ohjelman vakioiden ja "makrofunktioiden" määrittelyyn [aliluku 4.3 sivulla 105].

*Nimetyt vakiot toteutetaan C++:ssa **const**-muuttujina ja makrot korvataan **inline**-funktioilla.*

4.6. Ohjelmakoodin toimintaa ohjaavat raja-arvot ja muut vakiot määritellään keskitetysti yhdessä paikassa vakiomuuttujiksi (numeeristen vakioiden käyttö ohjelmassa on kiellettyä). Lyhyet merkitykseltään varmasti pysyvät vakiot kuten esimerkiksi nollaosoitin (`0`) tai "kymmenen alkion silmukka" (`-5 ... 5`) on kuitenkin usein selkeämpää kirjoittaa suoraan käyttöpaikassa näkyviin.

Kaikki numeeriset arvot tulisi määritellä yhdessä keskitetyssä paikassa (otsikkotiedosto) selkeästi nimetyiksi vakiomuuttujiksi, joita käytetään muualla ohjelmakoodissa. Vakioiden arvojen muuttaminen myöhemmin on tällöin helppoa. Näin tulisi tehdä kaikille arvoille, joita mahdollisesti halutaan muuttaa ohjelman jatkokehityksessä. Jos tiedetään hyvin varmaksi, ettei jokin arvo muutu (nolla tai "tässä laiteohjaimessa on kymmenen alkion alustus"), niin arvon kirjoittaminen ilman vakiomuuttujan tuomaa epäsuoruutta on perusteltua.

4.7. Totuusarvot ilmaistaan ISOC++:n tietotyypillä **bool**, jonka mahdolliset arvot ovat **false** ja **true**.

C-kielen "totuusarvot" nolla ja nollasta poikkeavat kokonaisluvut ovat edelleen toimivia, mutta niiden käyttöä ei suositella ISOC++:ssa.

4.8. Kun tyyppi määritellään vakioksi, **const**-sanan paikka tyypin määrittelyssä on itse tyyppinimen *jälkeen*. Tämä suositeltava tyyli on yleistymässä uusissa ohjelmissa. Aiemmin koodissa on ollut tapana kirjoittaa **const** *ennen* tyyppinimeä. Tärkeintä on, että käytäntö on yhtenäinen koko ohjelmassa. [aliluku 4.3 sivulla 105]

```
int const VAKIO = 3;       // Kokonaislukuvakio
int const* ptr = &VAKIO; // vakio-osoitin eli osoitin vakioon
int* const PTR2 = 0;       // Osoitinvakio jota ei saa muuttaa
```

5. **Asemointi ja tyyli**

5.1. Yli tuhannen rivin tiedostojen käsittely on hankalaa, joten niitä tulisi välttää.

5.2. Yhden (jäsen)funktion toteutuksen pituuden ei tulisi ylittää sataa riviä.

5.3. Päätelaitteiden käytännöksi on muodostunut 80 merkkiä leveä näyttö. Tästä johtuen yli 79 merkin pituiset rivit kannattaa jakaa useammalle riville.

```
std::string const esimerkkijono = "Katenointi "
                                  "on ominaisuus jolla kääntäjä"
                                  " yhdistää peräkkäin olevat"
                                  " merkkijonoliteraalit yhdeksi.\n";

std::vector<unsigned long int>
laskeTaulukkoVastaus( unsigned int pituus,
                      std::map<unsigned long int,
                      std::string> const& tiedot );
```

5.4. Osoittimen ja viitteen merkki kuuluvat tyyppinimen yhteyteen.

```
Paivays* ptr = NULL;
void tulostaPaivays( Paivays& paivaysOlio );
```

5.5. Primitiivejä **`if`**, **`else`**, **`while`**, **`for`** ja **`do`** seuraa aina koodilohko, joka on sisennetty johdonmukaisesti välilyöntimerkeillä.

Sarkaimen (tabulator) käyttö ei ole suositeltavaa, koska erilaiset tekstieditorit ja katseluohjelmat näyttävät sen sisennyksen eri tavoin ja voivat tuottaa lukukelvottoman lopputuloksen koodista, joka alkuperäisessä kirjoitusympäristössä on näyttänyt selkeältä.

5.6. Koodilohkon alku- ({) ja loppumerkki (}) sijoitetaan omalle rivilleen samalle sarakkeelle (tai jollakin muulla yhtenäisellä ja selkeällä tavalla eroteltuna muusta koodista).

5.7. Jokaisessa **`switch`**-lauseessa on **`default`**-määre.

5.8. Jokainen **switch**-lauseen **case**-määreen toteutus on oma koodilohkonsa ja se päättyy lauseeseen **break** (samalla toteutuksella voi olla useita **case**-määreitä).

```
switch( merkki )
{
  case 'a':
  case 'A':
  {
      ⋮
    break;
  }

  default:
  {
    break;
  }
}
```

6. C-kielen rakenteet C++:ssa

6.1. Ohjelmasta ei saa poistua funktiolla `exit()` tai `abort()`.

*Nämä funktiot lopettavat koko ohjelman suorituksen ilman, että ohjelman kaikki oliot tuhoutuvat hallitusti (niiden purkajat jäävät suorittamatta). Oikea tapa on aina poistua funktiosta `main()` normaalisti (**return**-lauseella).*

6.2. `main()`-funktion paluuarvo on aina **int**.

ISOC++ -standardin määrittelemä ainoa mahdollinen paluuarvo.

6.3. `main()`-funktiota ei saa kutsua, siitä ei saa ottaa funktio-osoitinta, sitä (sen nimeä) ei saa kuormittaa eikä se saa olla **inline** eikä staattinen funktio — toisin sanoen funktionimeä `main` ei saa käyttää kuten tavallista funktiota.

ISOC++ -standardi käsittelee `main()`-funktion erikoistapauksena (se ei siis ole tavallinen funktio), ja kieltää nämä käyttötavat.

6.4. **goto**-lausetta ei saa käyttää.

Ohjelman normaalin toiminnan hallintaan on olemassa parempia kontrollirakenteita ja virhetilanteissa “hyppääminen muualle” toteutetaan poikkeusten avulla.

6.5. Käytä tietovirtoja (`cin`, `cout`, `cerr` ja `clog`) C-kielen IO-funktioiden sijaan.

Tietovirtojen operaatiot tekevät tyyppitarkistuksia, joita esimerkiksi C-funktio `printf()` ei tee.

6.6. Funktioita `malloc()`, `realloc()` ja `free()` ei saa käyttää, vaan niiden sijasta käytetään operaattoreita **`new`** ja **`delete`** [aliluku 3.2 sivulla 71] [aliluku 3.5 sivulla 85].

Uudet operaattorit kutsuvat olioiden dynaamisen käsittelyn yhteydessä rakentajia ja purkajia, joista vanhat muistinvarausrutiinit eivät tiedä mitään.

6.7. C:n kirjastoja käytettäessä niiden esittelyt tulee ottaa käyttöön määreellä **`extern`** `"C"`.

Vain tällä merkinnällä varmistetaan yhteensopivuus C-kielisen kirjaston kutsumekanismin kanssa.

```
extern "C" {
#include <gtk/gtk.h>
#include <curses.h>
}
```

7. Tyyppimuunnokset

7.1. Mikään osa ohjelmakoodista ei saa luottaa implisiittiseen tyyppimuunnokseen.

7.2. Ohjelmassa tulee käyttää kielen uusia tyyppimuunnosoperaattoreita (**`static_cast`**, **`dynamic_cast`** ja **`reinterpret_cast`**) vanhempien tyyppimuunnosten sijaan [aliluku 7.4.1 sivulla 228].

7.3. Vakio-ominaisuutta ei saa poistaa tyyppimuunnoksella (Operaattoria **`const_cast`** ei saa käyttää) [aliluku 7.4.1 sivulla 228].

Vakio on suunnitteluvaiheessa määrätty ominaisuus, jota ei pitäisi olla tarvetta poistaa ohjelmointivaiheessa.

8. Funktiot ja jäsenfunktiot

8.1. Funktion parametrien nimet on annettava sekä esittelyssä että määrittelyssä ja niiden on oltava molemmissa samat.

8.2. Funktion olioparametrin välitykseen käytetään (vakio)viitettä aina kun se on mahdollista [aliluku 4.3 sivulla 105] [aliluku 7.3 sivulla 224].

8.3. Funktiolla ei saa olla määrittelemätöntä parametrilistaa (ellipsis notation).

Määrittelemätön parametrilista poistaa käytöstä kaikki kääntäjän tekemät tarkistukset parametrinvälityksessä.

8.4. Funktion mahdolliset oletusparametrit ovat näkyvissä esittelyssä eikä niitä saa lisätä määrittelyssä.

8.5. Funktion paluuarvo on aina määriteltävä.

8.6. Funktio ei saa palauttaa osoitinta tai viitettä sen omaan paikalliseen dataan.

Paikallinen data ei ole kielen määrittelyn mukaan enää käytettävissä kun funktiosta on palattu.

8.7. Julkisen rajapinnan jäsenfunktio ei saa palauttaa olion jäsenmuuttujaan viitettä eikä osoitinta (vakioviite ja vakioosoitin käy) [aliluku 4.2 sivulla 101].

8.8. Jäsenfunktiosta tehdään vakiojäsenfunktio aina kun se on mahdollista [aliluku 4.3 sivulla 105].

9. **Luokat ja oliot**

9.1. Oliot toteutetaan C++:n rakenteella **`class`** ja tietueet (esimerkiksi linkitetyn listan yksi alkio) toteutetaan rakenteella **`struct`**.

Näin erotellaan selkeästi tietueet (rakenteinen tietorakenne) olioista.

9.2. Tietueella saa olla rakentajia ja virtuaalinen toteutukseltaan tyhjä purkaja, muttei muita jäsenfunktioita.

Tietueissä säilytetään yhteenkuuluvaa tietoa, jolloin niiden ainoa sallittu toiminnallisuus liittyy uuden tietueen alustukseen ja tietueen tuhoutumiseen (virtuaalipurkaja tarvitaan periytetyn tietueen oikean tuhoutumisen varmistamiseksi).

9.3. Luokan osat **`public`**, **`protected`** ja **`private`** esitellään tässä järjestyksessä [aliluku 4.2 sivulla 101].

Luokan käyttäjää kiinnostaa ainoastaan julkisessa rajapinnassa olevat palvelut, joten niille selkein paikka on heti luokkaesittelyn alussa.

9.4. Luokan jäsenmuuttujat ovat aina näkyvyydeltään **`private`** [aliluku 4.2 sivulla 101].

9.5. Luokan esittelyssä ei koskaan ole toteutusta (toteutus mahdollisista **`inline`**-jäsenfunktioista on otsikkotiedostossa luokkaesittelyn jälkeen) [aliluku 2.3.3 sivulla 64].

Esittelyssä on nimensä mukaisesti tarkoitus ainoastaan esitellä jokin rakenne. Sen toteutusyksityiskohdat ovat muualla eikä rakenteen käyttäjän tarvitse niitä nähdä. Katso myös kohdat 2.3.4 ja 2.3.5.

9.6. Luokalla ei pääsääntöisesti saa olla ystäväfunktioita eikä ystäväluokkia (**friend**-ominaisuus) [aliluku 8.4 sivulla 258].

*"Ystävyys" rikkoo olioiden kapselointiperiaatetta vastaan. Yleinen poikkeus säännölle ovat kiinteästi yhteenkuuluvan luokkaryppään keskinäiset erioikeudet esimerkiksi saman ohjelmakomponentin sisällä. C++:ssa tulostusoperaattori **operator** << on pakko toteuttaa luokasta erillisenä funktiona, jolloin siitä usein tehdään ystäväfunktio (jos suunnittelussa tulostusta pidetään luokan julkisen rajapinnan operaationa).*

9.7. Luokalle kuormitettavien operaattoreiden semantiikka on säilytettävä (esimerkiksi **operator** + tekee aina yhteenlaskua "vastaavan" toiminnon).

10. Elinkaari

10.1. Muistinhallinta

10.1.1. Dynaamisesti varatun muistin vapauttaminen on pääsääntöisesti sen komponentin vastuulla (olio tai moduuli), joka varasi muistin [aliluku 3.5.4 sivulla 90].

10.1.2. Operaattorilla **delete** saa vapauttaa ainoastaan muistin, joka on varattu operaattorilla **new** (**delete**n parametri saa olla vain **new**:n palauttama osoitin tai arvo nolla) [aliluku 3.5.2 sivulla 89].

10.1.3. Operaattorilla **delete**[] saa vapauttaa ainoastaan muistin, joka on varattu operaattorilla **new**[] [aliluku 3.5.3 sivulla 90].

10.1.4. Jos **delete**-lauseen parametri on osoitinmuuttuja, johon voi sijoittaa, sen arvoksi sijoitetaan nolla **delete**ä seuraavassa lauseessa [aliluku 3.5.4 sivulla 90].

10.2. Olion luonti ja tuhoaminen

10.2.1. Dynaamisesti (operaattorilla **new**) luodun olion tuhoutuminen varmistetaan auto_ptr-rakenteella (tai muilla älykkäillä osoitintoteutuksilla) aina kun sen käyttö on

mahdollista [aliluku 3.5 sivulla 85] [aliluku 11.7 sivulla 383]. (Mutta esimerkiksi kohdan 12.3.4 sääntö estää `auto_ptr`:n taltioinnin STL:n säiliöihin.)

10.2.2. Jokaiselle luokalle on määriteltävä vähintään yksi rakentaja [aliluku 3.4.1 sivulla 80].

10.2.3. Rakentajissa ja purkajissa ei saa kutsua virtuaalifunktioita [aliluku 6.5.7 sivulla 170].

10.2.4. Rakentajissa ja purkajissa ei saa käyttää globaaleja eikä staattisia olioita.

Olio voi itse olla staattinen tai globaali eivätkä sen käyttämät muut vastaavan tason oliot ole vielä välttämättä olemassa alustettuna kun rakentajan suoritus alkaa. Vastaavasti purkajassa voidaan vahingossa viitata jo aiemmin tuhottuun olioon.

10.2.5. Jos rakentajan toteutuksessa (toteutustiedostossa) tulostetaan, ennen luokan esittelyä (otsikkotiedosto) pitää ottaa käyttöön otsikkotiedosto <`iostream`>.

Ainoastaan tällä tavalla voidaan varmistua siitä, että tulostusolio (esim. `cout`) on olemassa ennen luokasta tehtyä oliota.

10.2.6. Yksiparametrinen rakentaja on merkittävä **`explicit`**-määreellä [aliluku 7.4.2 sivulla 232], ellei se erityisesti ole tarkoitettu implisiittiseksi tyyppikonversioksi. ***Poikkeuksena kopiorakentaja, jolle tätä määrettä ei saa laittaa.***

10.2.7. Itse tehty kantaluokan purkaja on aina **`public`** ja virtuaalinen tai **`protected`** ja ei-virtuaalinen. [aliluku 6.5.5 sivulla 168].

10.2.8. Purkajan tulee vapauttaa kaikki tuhoutumishetkellä olion vastuulla olevat resurssit [aliluku 3.4.2 sivulla 83].

10.3. Olion kopiointi

10.3.1. Luokalle esitellään aina kopiorakentaja [aliluku 7.1.2 sivulla 206].

10.3.2. Tarpeettoman kopiorakentajan käyttö estetään esittelemällä se ilman toteutusta luokan **`private`**-rajapintaan [aliluku 7.1.2 sivulla 206].

10.3.3. Periytetyn luokan kopiorakentajan tulee kutsua kantaluokan kopiorakentajaa alustuslistassaan [aliluku 7.1.2 sivulla 206].

10.4. Olion sijoitus

10.4.1. Jokaisessa luokassa esitellään oma sijoitusoperaattori [aliluku 7.2.2 sivulla 216].

10.4.2. Tarpeettoman sijoitusoperaattorin käyttö estetään esittelemällä se ilman toteutusta luokan `private`-osassa [aliluku 7.2.2 sivulla 216].

10.4.3. Periytetyn luokan sijoitusoperaattorin tulee kutsua kantaluokan sijoitusoperaattoria [aliluku 7.2.2 sivulla 216].

10.4.4. Sijoitusoperaattorin tulee aina varautua "itseen sijoitamista" (`x = x`) vastaan [aliluku 7.2.2 sivulla 216].

10.4.5. Sijoitusoperaattorin paluuarvo on `*this` [aliluku 7.2.2 sivulla 216].

11. **Periytyminen**

11.1. Ainoastaan `public`-periytymisen käyttö on sallittua mallintamaan olio-ohjelmoinnin periytymistä [aliluku 6.3 sivulla 149].

11.2. Periytyminen on staattinen "is-a"-suhde. "has-a" toteutetaan jäsenmuuttujilla ja tyyppiriippumaton koodi malleilla [aliluku 6.3.4 sivulla 155].

11.3. Moniperiytymistä saa käyttää ainoastaan rajapinnan tai toiminnallisen yksikön periyttämiseen (interface ja mixing) [aliluku 6.3 sivulla 149].

Moniperiytyminen on monimutkainen ominaisuus, jota harvoin käytetään oliosuunnittelussa. Rajapintojen periyttäminen taas on paljon käytetty osa oliosuunnittelua.

11.4. Rajapintaluokan kaikki jäsenfunktiot ovat puhtaita virtuaalifunktioita (pure virtual) [aliluku 6.6 sivulla 172].

11.5. Periytetyn luokan rakentajan alustuslistassa täytyy kutsua kantaluokan rakentajaa [aliluku 6.3.2 sivulla 152].

11.6. Kantaluokan määrittelemiin virtuaalifunktioihin, jotka peritetty luokka määrittelee uudelleen, liitetään avainsana

`virtual` myös periytetyssä luokassa [aliluku 6.5.5 sivulla 168].

11.7. Periytynyttä ei-virtuaalista jäsenfunktiota ei saa määritellä uudelleen [aliluku 6.5 sivulla 159].

11.8. Periytynyttä virtuaalisen jäsenfunktion oletusparametria ei saa määritellä uudelleen.

12. Mallit

12.1. Mallin tyyppiparametreilleen asettamat vaatimukset tulee dokumentoida mallin esittelyn yhteydessä [aliluku 9.5.4 sivulla 290].

12.2. Mallin käyttäjille tulisi tarvittaessa tarjota valmiiksi optimoituja toteutuksia mallin erikoistusten avulla [aliluku 9.5.6 sivulla 295].

12.3. STL

12.3.1. C-kielen vanhan taulukkotyypin sijaan tulee käyttää STL:n sarjasäiliöitä `vector` tai `deque` [aliluku 10.2 sivulla 310] [aliluku A.3 sivulla 414].

Uudet rakenteet ovat helppokäyttöisempiä. Jos jokin (vanhan ohjelmakoodin) funktio tarvitsee esimerkiksi parametrikseen taulukon, sille voidaan antaa osoitin vektorin alkuun (`&v[0]`).

12.3.2. STL:n sisäisestä toiminnasta ei saa tehdä oletuksia. Ainoastaan ISOC++:n määrittelemiä ominaisuuksia saa hyödyntää.

12.3.3. Käytetyille STL-säiliöille annetaan kuvaavat nimet tyyppialiaksen avulla.

```
typedef std::map<std::string,unsigned int>
        Puhelinluettelo;
```

12.3.4. STL:n säiliöön saa tallettaa ainoastaan olioita, joilla on kopiorakentaja ja sijoitusoperaattori, jotka tuottavat uuden tilaltaan identtisen olion [aliluku 10.2 sivulla 310].

12.3.5. Käytettäessä STL:n säiliöitä on varmistuttava siitä, että käytettyjen operaatioiden pahimman tapauksen ajankäyttö on riittävän tehokasta ohjelmiston toiminnan kannalta [aliluku 10.1 sivulla 303].

12.3.6. STL:n säiliöiden yhteydessä tulee käyttää STL:n algoritmeja omien toteutusten sijaan [aliluku 10.4 sivulla 337].

STL:n algoritmit ovat luultavasti omia toteutuksia tehokkaampia ja ne pystyvät tekemään sisäisiä optimointeja.

12.3.7. STL:n säiliön alkioiden läpikäynti toteutetaan iteraattoreiden avulla (`begin()`–`end()` -väli) [aliluku 10.3 sivulla 326].

Esimerkki funktiosta, joka käy läpi kaikki säiliön alkiot riippumatta siitä mitä STL:n säiliöistä on käytetty parametrin toteutukseen:

```
void tulostaPuhelinluettelo(Puhelinluettelo& tk)
{
  Puhelinluettelo::const_iterator alkio;
  for (alkio = tk.begin(); alkio != tk.end(); ++alkio)
  // Vertailu "alkio < tk.end()" ei toimi
  // kaikkilla säiliöillä (niiden iteraattoreilla)
  {
    std::cout << *alkio << std::endl;
  }
}
```

12.3.8. Iteraattorin lisääminen ja vähentäminen suoritetaan etuliiteoperaattorilla (preincrement ja predecrement) [aliluku 10.3 sivulla 326].

Lausekkeen evaluoinnin jälkeen suoritettava operaatio (postincrement) on tehottomampi, koska se joutuu palauttamaan kopion iteraattorin vanhasta arvosta.

12.3.9. Mitätöitynyttä iteraattoria ei saa käyttää (ainoastaan tuhoaminen ja uuden arvon sijoittaminen ovat sallittuja operaatioita) [aliluku 10.3 sivulla 326].

13. **Poikkeukset**

13.1. Poikkeuksia saa käyttää ainoastaan virhetilanteiden ilmaisemiseen [aliluku 11.4 sivulla 374].

Poikkeusten käsittely on muita kontrollirakenteita raskaampaa, joten niitä ei kannata käyttää ohjelmiston normaalin kontrollin ohjaamiseen.

13.2. Jos poikkeuskäsittelijä ei pysty täysin toipumaan sieppaamastaan poikkeuksesta, se "heitetään" uudelleen ylemmälle tasolle [aliluku 11.4.2 sivulla 377].

13.3. Pääohjelmasta ei saa vuotaa poikkeuksia ulos [aliluku 11.4.2 sivulla 377].

Useat ajoympäristöt antavat erittäin epäselviä ilmoituksia ohjelmasta ulos vuotavien poikkeusten yhteydessä.

13.4. Luokan purkajasta ei saa vuotaa poikkeuksia [aliluku 11.5 sivulla 380].

13.5. Pääsääntöisesti funktioiden poikkeusmäärelistaa ei saa käyttää. Vain jos funktion kaikissa käyttötilanteissa varmasti tiedetään siitä mahdollisesti vuotavat poikkeukset, voi poikkeukset luetella funktioesittelyn poikkeuslistassa. [aliluku 11.6 sivulla 382].

Kirjallisuutta

[Abadi ja Cardelli, 1996] Martín Abadi ja Luca Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer-Verlag, 1996. ISBN 0-387-94775-2.

[Abrahams, 2003] David Abrahams. Exception-Safety in Generic Components. `http://www.boost.org/more/generic_exception_safety.html`, toukokuu 2003.

[Adams, 1991] James Adams. *Insinöörin maailma*. Art House Oy, 1991. ISBN 951-884-163-2. Suomentanut Kimmo Pietiläinen.

[Aikio ja Vornanen, 1992] Annukka Aikio ja Rauni Vornanen, toim. *Uusi sivistyssanakirja*. Otava, 11. painos, 1992. ISBN 951-1-11365-8.

[Alexandrescu, 2001] Andrei Alexandrescu. *Modern C++ Design*. C++ In-Depth Series. Addison-Wesley, 2001. ISBN 0-201-70431-5. `http://www.moderncppdesign.com/`.

[Alexandrescu, 2003] Andrei Alexandrescu. Modern C++ Design, Chapter 7 — Smart Pointers. `http://www.aw.com/samplechapter/0201704315.pdf`, toukokuu 2003. Kirjan [Alexandrescu, 2001] luku 7 verkossa julkaistuna.

[Arnold ja Gosling, 1996] Ken Arnold ja James Gosling. *The Java Programming Language*. Addison-Wesley, 1996. ISBN 0-201-63455-4.

[Beck ja Cunningham, 1989] Kent Beck ja Ward Cunningham. A Laboratory for Teaching Object-Oriented Thinking. Teoksessa *OOPSLA'89 – Conference on Object-Oriented Programming Systems, Languages and Applications*. ACM SIGPLAN, 1989.

[Binder, 1999] Robert V. Binder. The Percolation Pattern – Techniques for Implementing Design by Contract in C++. *C++ Report*, sivut 38–44, toukokuu 1999.

[Booch *ja muut*, 1999] Grady Booch, James Rumbaugh ja Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999. ISBN 0-201-57168-4.

[Booch, 1987] Grady Booch. *Software Engineering with Ada, 2nd edition*. The Benjamin/Cummings Publishing Company, 1987. ISBN 0-8053-0604-8.

[Booch, 1991] Grady Booch. *Object-Oriented Design with Applications*. The Benjamin/Cummings Publishing Company, 1991. ISBN 0-8053-0091-0.

[Boost, 2003] C++ Boost library. `http://www.boost.org/`, helmikuu 2003.

[Böszörményi ja Weich, 1996] László Böszörményi ja Carsten Weich. *Programming in Modula-3 → An Introduction in Programming with Style*. Springer-Verlag, 1996. ISBN 3-540-57912-5.

[Budd, 2002] Timothy Budd. *An Introduction to Object-oriented Programming, 3rd edition*. Addison-Wesley, 2002. ISBN 0-201-76031-2.

[Carroll, 1865] Lewis Carroll. Alice's Adventures in Wonderland. Kirjassa *The Complete Works of Lewis Carroll* [1988].

[Carroll, 1988] Lewis Carroll. *The Complete Works of Lewis Carroll*. The Penguin Group, 1988. ISBN 0-14-010542-5.

[Clements ja Parnas, 1986] Paul Clements ja David Parnas. A Rational Design Process: How and Why to Fake It. *IEEE Transactions on Software Engineering*, 12(2):251–257, helmikuu 1986.

[Coplien, 1992] James O. Coplien. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, 1992. ISBN 0-201-54855-0.

[Coplien, 1999] James O. Coplien. *Multi-Paradigm Design for C++*. Addison-Wesley, 1999. ISBN 0-201-82467-1.

[Coplien, 2000] James Coplien. C++ idioms patterns. Teoksessa Neil Harrison, Brian Foote ja Hans Rohnert, toim., *Pattern Languages of Program Design 4*, luku *10*, sivut 167–197. Addison-Wesley, 2000. ISBN 0-201-43304-4.

[Cormen *ja muut*, 1990] Thomas H. Cormen, Charles E. Leiserson ja Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990. ISBN 0-262-53091-0.

[Czarnecki ja Eisenecker, 2000] Krysztof Czarnecki ja Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000. ISBN 0-201-30977-7.

[Dijkstra, 1972] Edsger W. Dijkstra. The Humble Programmer (1972 Turing Award Lecture). *Communications of the ACM (CACM)*, 15(10):859–866, lokakuu 1972.

[Doxygen, 2005] Doxygen homepage. `http://www.doxygen.org/`, huhtikuu 2005.

[Driesen ja Hölzle, 1996] Karel Driesen ja Urz Hölzle. The Direct Cost of Virtual Function Calls in C++. Teoksessa Carrie Wilpolt, toim., *OOPSLA'96 – Conference on Object-Oriented Programming Systems, Languages and Applications*. ACM SIGPLAN, Addison-Wesley, lokakuu 1996.

[Egan, 1995] Greg Egan. *Axiomatic*. Millennium, 1995. ISBN 1-85798-416-1.

[Fomitchev, 2001] Boris Fomitchev. STLport. STLport home page `http://www.stlport.org/`, elokuu 2001.

[Gaiman *ja muut*, 1994] Neil Gaiman, Jill Thompson ja Vince Locke. *Brief Lives*. Sandman-albumi. DC Comics, 1994. ISBN 1-56389-137-9.

[Gamma *ja muut*, 1995] Erich Gamma, Richard Helm, Ralph Johnson ja John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. ISBN 0-201-63361-2.

[Gillam, 1997] Richard Gillam. The Anatomy of the Assignment Operator. *C++ Report*, sivut 15–23, marraskuu–joulukuu 1997.

[Haikala ja Märijärvi, 2002] Ilkka Haikala ja Jukka Märijärvi. *Ohjelmistotuotanto*. Suomen ATK-kustannus Oy, 2002. ISBN 952-14-0486-8.

[Heller, 1961] Joseph Heller. CATCH-22. Gummerus, 1961. ISBN 951-20-4558-3. Suomentanut Markku Lahtela.

[ISO, 1998] ISO/IEC. *International Standard 14882 – Programming Languages – C++*, 1. painos, syyskuu 1998.

[Jacobson *ja muut*, 1994] I. Jacobson, M. Christerson, P. Johnsson ja G. Övergaard. *Object Oriented Software Engineering, A Use Case Driven Approach (Revised Printing)*. Addison-Wesley, 1994. ISBN 0-201-54435-0.

[JavaBeans, 2001] JavaBeans Component Architecture. `http://java.sun.com/beans/`, elokuu 2001.

[Josuttis, 1999] Nicolai M. Josuttis. *The C++ Standard Library*. Addison-Wesley, 1999. ISBN 0-201-37926-0.

[Kerninghan ja Ritchie, 1988] Brian W. Kerninghan ja Dennis M. Ritchie. *The C Programming Language, 2nd edition*. Prentice Hall Software Series, 1988. ISBN 0-13-110370-9.

[Koenig ja Moo, 2000] Andrew Koenig ja Barbara Moo. *Accelerated C++*. Addison-Wesley, 2000. ISBN 0-201-70353-X.

[Koskimies, 2000] Kai Koskimies. *Oliokirja*. Suomen ATK-kustannus Oy, 2000. ISBN 951-762-720-3.

[Krohn, 1992] Leena Krohn. *Matemaattisia olioita*. WSOY, 1992. ISBN 951-0-18407-1.

[Lem, 1965] Stanislaw Lem. *Kyberias*, luku *Ensimmäinen matka (A) eli Trurlin elektrubaduuri*. Kirjayhtymä, 1965. ISBN 951-26-2955-0. Suomentanut Matti Kannosto.

[Liberty, 1998] Jesse Liberty. *Beginning Object-Oriented Analysis and Design with C++*. Wrox Press Ltd, 1998. ISBN 1-861001-33-9.

[Linnaeus, 1748] Carl Linnaeus. *Systema Naturae*. Holmiae, 1748.

[Lions, 1996] Prof. J. L. Lions. ARIANE 5 flight 501 failure. Raportti, Inquiry Board, heinäkuu 1996. http://www.cafm.sbu.ac.uk/cs/people/jpb/teaching/ethics/ariane5anot.html.

[Lippman ja Lajoie, 1997] Stanley B. Lippman ja Josée Lajoie. *C++ Primer 3rd edition*. Addison-Wesley, 1997. ISBN 0-201-82470-1.

[Lippman, 1996] Stanley B. Lippman. *Inside the C++ Object Model*. Addison-Wesley, 1996. ISBN 0-201-83454-5.

[McConnell, 1993] Steve McConnell. *Code Complete*. Microsoft Press, 1993. ISBN 1-55615-484-4.

[Meyer, 1997] Bertrand Meyer. *Object-Oriented Software Construction, 2nd edition*. Prentice Hall, 1997. ISBN 0-13-629155-4.

[Meyers, 1996] Scott Meyers. *More Effective C++*. Addison-Wesley, 1996. ISBN 0-201-63371-X. Saatavilla elektronisessa muodossa osana teosta [Meyers, 1999].

[Meyers, 1998] Scott Meyers. *Effective C++ 2nd edition*. Addison-Wesley, 1998. ISBN 0-201-92488-9. Saatavilla elektronisessa muodossa osana teosta [Meyers, 1999].

[Meyers, 1999] Scott Meyers. *Effective C++ CD*. Addison-Wesley, 1999. ISBN 0-201-31015-5. CD:llä julkaistu elektroninen kirja.

[OMG, 2002a] OMG. UML Resource Page. http://www.omg.org/technology/uml/, syyskuu 2002.

[OMG, 2002b] OMG. What Is OMG-UML and Why Is It Important? OMG Press Release, http://www.omg.org/gettingstarted/what_is_uml.htm, elokuu 2002.

[Pirsig, 1974] Robert M. Pirsig. *ZEN ja moottoripyörän kunnossapito*. WSOY, 1974. ISBN 951-0-19300-3. Suomentanut Leena Tamminen.

[Pratchett, 1987] Terry Pratchett. *Equal Rites*. Corgi, 1987. ISBN 0-552-13105-9.

[Roszak, 1992] Theodore Roszak. *Konetiedon kritiikki*. Art House Oy, 1992. ISBN 951-884-085-7. Suomentanut Maarit Tillman.

[Rumbaugh *ja muut*, 1991] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy ja William Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991. ISBN 0-13-630054-5.

[Rumbaugh *ja muut*, 1999] James Rumbaugh, Ivar Jacobson ja Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999. ISBN 0-201-30998-X.

[Sethi, 1996] Ravi Sethi. *Programming Languages, Concepts & Constructs, 2nd edition*. Addison-Wesley, 1996. ISBN 0-201-59065-4.

[SETI, 2001] SETI Newsletter: Result verification. `http://setiathome.ssl.berkeley.edu/newsletters/newsletter8.html`, heinäkuu 2001.

[Shakespeare, 1593] William Shakespeare. The Tragedy of Titus Andronicus. Kirjassa *The Complete Works of William Shakespeare* [1994]. Vuosiluku summittainen.

[Shakespeare, 1994] William Shakespeare. *The Complete Works of William Shakespeare*. Wordsworth Editions Ltd, 1994. ISBN 1-85326-810-0.

[Stroustrup, 1994] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1994. ISBN 0-201-54330-3.

[Stroustrup, 1997] Bjarne Stroustrup. *The C++ Programming Language 3rd edition*. Addison-Wesley, 1997. ISBN 0-201-88954-4. Teoksesta on myös suomennos [Stroustrup, 2000].

[Stroustrup, 2000] Bjarne Stroustrup. *C++-ohjelmointi*. Teknolit, 2000. ISBN 951-846-026-4. Suomennos teoksesta [Stroustrup, 1997].

[Sudo, 1999] Philip Toshio Sudo. *ZEN Computer*. Simon & Schuster New York, 1999. ISBN 0-684-85410-4.

[Sun Microsystems, 2005] Sun Microsystems. Javadoc tool home page. `http://java.sun.com/j2se/javadoc/`, huhtikuu 2005.

[Sutter, 1999] Herb Sutter. When Is a Container Not a Container? *C++ Report*, 11(5):60–64, toukokuu 1999.

[Sutter, 2000] Herb Sutter. *Exceptional C++*. C++ In-Depth Series. Addison-Wesley, 2000. ISBN 0-201-61562-2.

[Sutter, 2002a] Herb Sutter. A Pragmatic Look at Exception Specifications – The C++ feature that wasn't. *C/C++ Users Journal*, 20(7):59–64, heinäkuu 2002.

[Sutter, 2002b] Herb Sutter. "Export" Restrictions, Part 1. *C/C++ Users Journal*, 20(9):50–55, syyskuu 2002.

[Sutter, 2002c] Herb Sutter. *More Exceptional C++*. C++ In-Depth Series. Addison-Wesley, 2002. ISBN 0-201-70434-X.

[Sutter, 2003] Herb Sutter. Exception Safety and Exception Specifications: Are They Worth It? `http://www.gotw.ca/gotw/082.htm`, maaliskuu 2003.

[The Doors FAQ, 2002] THE DOORS Frequently Asked Questions (FAQ). `http://classicrock.about.com/library/artists/bldoors_faq.htm`, marraskuu 2002.

[Vandevoorde ja Josuttis, 2003] David Vandevoorde ja Nicolai M. Josuttis. *C++ Templates — The Complete Guide*. Addison-Wesley, 2003. ISBN 0-201-73484-2.

[Wikström, 1987] Åke Wikström. *Functional Programming Using Standard ML*. Prentice Hall, 1987. ISBN 0-13-331661-0.

[Wilkinson, 1995] Nancy M. Wilkinson. *Using CRC Cards, An Informal Approach to Object-Oriented Development*. SIGS BOOKS, 1995. ISBN 0-13-374679-8.

Englanninkieliset termit

English	Suomi
abstract base class	abstrakti kantaluokka, s. 144
actor	käyttäjärooli, s. 123
adaptive system	mukautuva järjestelmä, s. 351
aggregate	kooste, s. 134
allocator	varain, s. 305
ambiguous	moniselitteinen, s. 181
amortized constant complexity	amortisoidusti vakioaikainen tehokkuus, s. 310
ancestor	esi-isä, s. 144
assignment operator	sijoitusoperaattori, s. 216
associative container	assosiatiivinen säiliö, s. 317
attribute	attribuutti, s. 122
base class	kantaluokka, s. 144
basic (exception) guarantee	perustakuu, s. 390
bidirectional iterator	kaksisuuntainen iteraattori, s. 331
bitset	bittivektori, s. 325
bottom-up	kokoava jaottelu, s. 118
bridge	silta, s. 274
call by value	arvonvälitys, s. 224
call-through function	läpikutsufunktio, s. 180
catch (exceptions)	siepata, s. 374
class	luokka, s. 60

class definition	luokan esittely, s. 61
class hierarchy	luokkahierarkia, s. 144
class invariant	luokkainvariantti, s. 246
class object	luokkaolio, s. 249
class template	luokkamalli, s. 287
class template partial specialization	luokkamallin osittaiserikoistus, s. 297
closure	sulkeuma, s. 343
commonality and variability analysis	pysyvyys- ja vaihtelevuusanalyysi, s. 264
compile-time complexity	käännösaikainen tehokkuus, s. 310
component	komponentti, s. 38
composite	kokoelma, s. 271
composite aggregate	muodostuminen (UML), s. 134
const	vakio, s. 105
constant complexity	vakioaikainen tehokkuus, s. 310
constructor	rakentaja, s. 80
container	säiliö, s. 304
container adaptor	säiliösovitin, s. 325
conversion member function	muunnosjäsenfunktio, s. 236
copy constructor	kopiorakentaja, s. 206
cursor	kohdistin, s. 273
data member	jäsenmuuttuja, s. 61
deep copy	syväkopiointi, s. 205
default constructor	oletusrakentaja, s. 82
derived class	aliluokka, s. 144
descendant	jälkeläinen, s. 144
Design By Contract	sopimussuunnittelu, s. 240
design pattern	suunnittelumalli, s. 267
destructor	purkaja, s. 83
double-ended queue	pakka, s. 315

dynamic binding .... dynaaminen sitominen, s. 161

encapsulation .... kapselointi, s. 33
exception .... poikkeus, s. 367
exception guarantee .... poikkeustakuu, s. 389
exception handler .... poikkeuskäsittelijä, s. 374
exception neutrality .... poikkeusneutraalius, s. 392
exception specification .... poikkeusmääre, s. 382

forward declaration .... ennakkoesittely, s. 112
forward iterator .... eteenpäin-iteraattori, s. 331
framework .... sovelluskehys, s. 200
function object .... funktio-olio, s. 340
function object adaptor .... funktio-oliosovitin, s. 347
function pointer .... funktio-osoitin, s. 341
function template .... funktiomalli, s. 286
function try block .... funktion valvontalohko, s. 396
functor .... funktio-olio, s. 340

garbage collection .... roskienkeruu, s. 71
generalization .... yleistäminen, s. 147
generic algorithm .... geneerinen algoritmi, s. 304
generic programming ..... geneerinen ohjelmointi, s. 291
genericity .... yleiskäyttöisyys, s. 264
getter .... anna-jäsenfunktio, s. 103

header .... otsikkotiedosto, s. 39
hide .... peittää, s. 168

identity .... identiteetti, s. 58
inheritance .... periytyminen, s. 144
inheritance hierarchy ..... periytymishierarkia, s. 144
initialization list .... alustuslista, s. 81
input iterator .... syöttöiteraattori, s. 329

insert iterator	lisäys-iteraattori, s. 336
inserter	lisäys-iteraattori, s. 336
instantiation	instantiointi, s. 285
interface class	rajapintaluokka, s. 190
invalid iterator	kelvoton iteraattori, s. 334
invalidate (iterator)	mitätöidä, s. 334
invariant	invariantti, s. 243
iterator	iteraattori, s. 304
iterator adaptor	iteraattorisovitin, s. 336
key	avain, s. 317
linear complexity	lineaarinen tehokkuus, s. 310
list	lista, s. 315
logarithmic complexity	logaritminen tehokkuus, s. 310
map	assosiaatiotaulu, s. 321
member function	jäsenfunktio, s. 64
member function template	jäsenfunktiomalli, s. 289
metaclass	metaluokka, s. 249
metafunction	metafunktio, s. 352
metaprogram	metaohjelma, s. 349
metaprogramming	metaohjelmointi, s. 349
method	metodi, s. 64
minimal (exception) guarantee	minimitakuu, s. 390
module	moduuli, s. 32
multimap	assosiaatiomonitaulu, s. 322
multiple inheritance	moniperiytyminen, s. 175
multiset	monijoukko, s. 319
namespace	nimiavaruus, s. 41
nothrow (exception) guarantee	nothrow-takuu, s. 392

order of growth . . . . kertaluokka, s. 309
output iterator . . . . tulostus-iteraattori, s. 330
overloading . . . . kuormittaminen, s. 81

parent class . . . . kantaluokka, s. 144
pattern language . . . . mallikieli, s. 268
postcondition . . . . jälkiehto, s. 241
precondition . . . . esiehto, s. 241
pure virtual function . . . . . . puhdas virtuaalifunktio, s. 172

quadratic complexity . . . . . . neliöllinen tehokkuus, s. 310

random access iterator . . . . hajasaanti-iteraattori, s. 331
range . . . . väli, s. 329
re-usability . . . . uudelleenkäytettävyys, s. 264
reference . . . . viite, s. 411
reference copy . . . . viitekopiointi, s. 203
reflection . . . . reflektio, s. 349
repeated multiple inheritance . . . . toistuva moniperiytyminen, s. 186
replicated multiple inheritance . . . . erotteleva moniperiytyminen, s. 186
responsibility . . . . vastuualue, s. 38
reverse iterator . . . . käänteis-iteraattori, s. 336

scope resolution operator . . näkyvyystarkenninoperaattori, s. 44
sequence . . . . sarja, s. 312
sequence diagram . . . . tapahtumasekvenssi, s. 138
set . . . . joukko, s. 318
setter . . . . aseta-jäsenfunktio, s. 103
shallow copy . . . . matalakopiointi, s. 204
shared aggregate . . . . jaettu kooste (UML), s. 134

shared multiple inheritance	yhdistävä moniperiytyminen, s. 187
slicing	viipaloituminen, s. 210
smart pointer	älykäs osoitin, s. 387
software crisis	ohjelmistokriisi, s. 27
specialization	erikoistaminen, s. 147
state machine	tilakone, s. 139
static data member	luokkamuuttuja, s. 250
static member function	luokkafunktio, s. 252
static metaprogramming ..	käännösaikainen metaohjelmointi, s. 350
stream iterator	virtaiteraattori, s. 336
strong (exception) guarantee	vahva takuu, s. 391
subclass	aliluokka, s. 144
superclass	kantaluokka, s. 144
template	malli, s. 283
template metaprogramming	template-metaohjelmointi, s. 350
template specialization	mallin erikoistus, s. 295
temporary object	väliaikaisolio, s. 225
throw (exceptions)	heittää, s. 374
top-down	osittava jaottelu, s. 118
try-block	valvontalohko, s. 374
type cast	tyyppimuunnos, s. 227
unnamed namespace	nimeämätön nimiavaruus, s. 49
use case	käyttötapaus, s. 123
valid iterator	kelvollinen iteraattori, s. 334
vector	vektori, s. 312
virtual function	virtuaalifunktio, s. 159

Hakemisto

:: 42
::Pari 289
::annaEka 289, 296, 297
::summaa 290
˜Liikkuva 195

abort 369
{ abstract } (UML) 144
abstrahoida **29**
abstrahointi 63
abstrakti **29**
abstrakti kantaluokka **144**, 172–175, 193
abstraktio **29**
abstraktiotaso 30, 202, 379
Adams, James 116
aikaaJaljella 104
ajoaikainen
  indeksointitarkastus 416
  jäsenfunktion valinta *katso* “dynaaminen sitominen”
  toteutuksen valinta 276
  tyyppitarkastus 164–167, 231, 284
  virheilmoitus 146, 303
<algorithm> 338
aliluokka **144**
alkuiteraattori 333
alustaminen
  jäsenmuuttujan 81
  olion *katso* “rakentaja”
Alustus 73
alustuslista **81**
  aliluokan rakentajan 153
amortisoidusti vakioaikainen tehokkuus **310**, 314
analysis paralysis 141
ankh 69
anna-jäsenfunktio 103
annaArvo 294
annaEka 288, 293, 296, 297
annaLuku 374
annaNimi 157
annaPaiva 107, 255, 257
annaPalautusPvm 262
arkkitehtuurimalli **267**
arvonvälitys *katso* “arvoparametri”, **224**, 291
arvoparametri 224–227
aseta-jäsenfunktio 103
Assert 246
assert 245
Assert_toteutus 247
assosiaatio **132–133**
assosiaatiomonitaulu (STL) **322**

assosiaatiotaulu (STL) **321–322**
assosiatiivinen säiliö (STL) **317–318**
at 243, 416
at 313, **313**, 315
attribuutti 55, 61, **122**, 126, 128
auto_ptr *katso* “automaattiosoitin”
automaattiosoitin **383–388**
  ja STL 387
  ja taulukot 387
  kopiointi 385–387
  omistaminen 384
  paluuarvona 385
  rajoitukset 387–388
  sijoitus 385–387
avain (STL) **317**

back 313, **313**, 315, 317, 326
back_inserter 336
bad_alloc 87
bad_cast 165
begin 435
begin 333
<bitset> 325
bitset 325
bittivektori (STL) **325**

Carroll, Lewis 275
<cassert> 245
catch 87, 395, 397
catch 375
catch (...) 378
CATCH-22 201
Cheshire cat 275
class 60, 285
clear 312, **313**
clone 205
commit or rollback 391
const 105
  -sanan paikka 106
const_cast 230–231
const_iterator 332
copy 338
copy (Smalltalk) 205
correctness formula 241
count 320
CRC-kortti **124–126**
<cstdlib> 46
<cstring> 46

Death 69
deck 315
deepCopy 206
delegointi **277**
delete 89–90
delete[] 90
<deque> 315
deque 315
Design By Contract 240
Dijkstra, Edsger W. 28
Dijkstra, Edsger Wybe 20
Doors **240**
double 237
Dreaded Diamond of Death 190
dualismi 58
dynaaminen elinkaari 79
  automaattiosoitin 384
  ja poikkeukset 380
  ja taulukot 90
  olion luominen 86–89
  olion luominen ja tuhoaminen 85–86
  olion tuhoaminen 89–90
  omistusvastuu 91

vaarat 86
virheiden välttäminen 90–94
dynaaminen muistinhallinta 85–86
dynaaminen sitominen 147, **160–164**
hinta 169–170
dynamic_cast 164, 231

ekskursio
periytyminen 136
poikkeukset 87
Elain::liiku 174
elektrubaduuri 26
elinkaari
C++ 76–79
dynaaminen 79
Java 74–75
Modula-3 72
olion 71–72
Smalltalk 74
staattinen 77–79
empty **313**, 325, 326
end 333
ennakkoesittely **112–113**
envelope/letter 275
equal_range 322
erase 312, **313**, 318
erikoisiteraattori 336
erikoistaminen **147**
erikoistus
funktiomallin 296
luokkamallin 296
luokkamallin osittais- 297
vektorin 324
erotteleva moniperiytyminen **186**
esi-isä **144**
esiehto **241**
esim 92
esim2 93
esittely
ennakko- 112
jäsenfunktio 64
luokka 60
estetiikka 116
eteenpäin-iteraattori **331**
exit 428
explicit 235
export 299

f 293
finalize (Java) 74
finally (Java) 380
find 318, 322, 339
flavours 178
flip 325
for 314, 415
for_each 339
friend 260
front 313, **313**, 315, 317, 326
front_inserter 336
funktio-olio 304, **343**, 340–348
kopiointi 346
funktio-oliosovitin **347**
funktio-osoitin **341**
funktiomalli **286–287**
erikoistus 296
funktion valvontalohko **396**
funktori *katso* “funktio-olio”

Gaiman, Neil 69
Gang of Four 268
geneerinen algoritmi 304–306, **337–340**
geneerinen ohjelmointi 291

geneerisyys 163, **263–348**
GoF-kirja 268

haeEnsimmainen 167
haeRaportinPaivays 425
Haikala, Ilkka 20, 28
hajasaanti-iteraattori **331**
hajotin *katso* “purkaja”
hamekangas 110
handle/body 275
Heller, Joseph 201
Henkilo 82, 394
Henkilo::Henkilo 82, 395, 397
Henkilotiedot 239
homesieni 144
hävitin *katso* “purkaja”

identiteetti **57–58**
idiomi **267**
if 217, 220, 223, 297, 399, 402, 404
imeta 192
indeksointi (STL) 313, 315, 321
inline 413–414
insert 312, **313**, 318
inserter 336
instanssimuuttuja *katso* “jäsenmuuttuja”
instantiointi 56
instantiointi, mallin **285**
interface
  Java 51
  Modula-3 51
  määrittely 31
  UML 128
« interface » (UML) 128
invariantti **243**
<iostream> 45
<iostream.h> 45
is-a 137, 155, 161
istream_iterator 336
iteraattori 304, **326–329**
  -kategoriat **329–332**
  -sovitin **336–337**, 337
  alku- 333
  eteenpäin- **331**
  hajasaanti- **331**
  ja algoritmit 337
  ja osoittimet 332
  ja säiliöt 332–334
  kaksisuuntainen **331**
  kelvollinen **334**
  kelvoton **334**
  käänteis- **336**
  lisäys- **336**
  loppu- 328, 333
  mitätöidä **334**
  suunnittelumalli **273–274**
  syöttö- **329**
  tulostus- **330**
  tyhjä 332
  vakio- 332
  virta- **336**
  väli **329**, 337
<iterator> 336
iterator 332

jaettu kooste 134
jarjestaJaPoista 340
JarjestettyTaulukko::EtsiJaMuutaAlkio 248
JarjestettyTaulukko::Invariantti 248
Java 51–53, 103, 204, 205, 249, 303, 411
jfunktio 66
Jim 240, **275**

johdettu luokka *katso* "aliluokka"
johtaminen *katso* "periytyminen"
jono (STL) **326**
joukko (STL) **318–319**
julkinen rajapinta **32**
jälkeläinen **144**
jälkiehto **241**
jäsenfunktio **64–66**
  anna- 103
  aseta- 103
  esittely 64
  kätkeminen 103
  muunnos- 236
  toteutus 64–66
  vakio- 107
jäsenfunktiomalli **289**
jäsenmuuttuja **61–64**
  alustaminen 81
  elinkaari 63
  kätkeminen 103
  nimeäminen 62
  olio 63
  osoitin 63
  rajapinnassa 102
  viite 63

kaksipäinen jono (STL) *katso* "pakka"
kaksisuuntainen iteraattori **331**
Kana::liiku 174
kantaluokka **144**
  abstrakti **144**, 172–175, 193
kantaluokkaosa 152
kantaluokkaosoitin 155
kantaluokkaviite 155
kapselointi **33**, 63, 113, 152, 256
  luokkien ystävyysominaisuus 260
karkauspäivä 110
kategorisointi 142
kauankoJouluun 68
kauankoPalautukseen 230
kayta 295
kaytakopiota 214
kaytto 114, 300
kaytto1 345
kaytto2 345
kelvollinen iteraattori **334**
kelvoton iteraattori **334**
kenttä *katso* "jäsenmuuttuja"
kerroPaivays 48
kertaluokka **309**
keskiarvo1 376
keskiarvo2 377
ketjusijoitus **217**
Kirja::˜Kirja 157, 404, 406, 409
Kirja::annaNimi 157
Kirja::annaPalautusPvm 104
Kirja::Kirja 157, 404, 406, 409
Kirja::sopiikoHakusana 161
Kirja::tulostaTiedot 161
Kirja::tulostaVirhe 161
Kirja::vaihda 408, 409
KirjaApu::tulostaTiedot 185
kirjanmerkki *katso* "iteraattori", 327
KirjastonKirja 158
KirjastonKirja::˜KirjastonKirja 158
KirjastonKirja::KirjastonKirja 158

KirjastonKirja::onkoMyohassa 158
KirjastonKirja::tulostaKirjatiedot 184, 185
KirjastonKirja::tulostaKTeostiedot 184, 185
KirjastonKirja::tulostaTiedot 162, 182
KirjastonTeosApu::tulostaTiedot 185
kloonaa 214
kloonaa-jäsenfunktio 213
kohdistin 273, 413
kokoava jaottelu **118**
kokoelma **271–273**
kokoonpanollinen kooste 134
komponentti **38**, 120, **121**
kontravarianssi 160
kooste **134–136**
  C++ 135
  koostuminen 135
  muodostuminen 134
  vs. periytyminen 198–199
koostuminen 135
kopioinnin estäminen 209–210
kopiointi **202–215**
  automaattiosoitin 385–387
  funktio-olio 346
  kloonaa-jäsenfunktio 213
  matala- **204–205**
  olion 202
  perusteet 202
  syvä- **205–206**
  viipaloituminen **210–215**
  viite- **203–204**
kopiorakentaja 83, **206–210**, 311
  estäminen 209–210
  ja periytyminen 207–208
  kääntäjän luoma 208–209
  poikkeusolion 374
kovarianssi 160
kuormittaminen **81**
kuukaudenAlkuun 225
kysyJaTestaa 345
käyttäjärooli **123**
käyttötapaus **123**, 125, 127, 140
käännösaikainen
  tyypitys 284
käännösaikainen
  metaohjelmointi **350**
  tehokkuus **310**
  tyypitys 303
käännösyksikkö 39
käänteis-iteraattori **336**
kääntäjän luoma
  kopiorakentaja 208–209
  oletusrakentaja 83
  oletusrakentajan kutsu 153
  sijoitusoperaattori 219–220

laatu 27, 120, *katso* [Pirsig, 1974]
LainausJarjestelma::LainausJarjestelma 134
laskeFibonacci 314
laskeKeskiarvo 376
laskeTaulukkoVastaus 427
Laskija::˜Laskija 254
Laskija::Laskija 254
Laskija::tulosta_tilasto 254
laula 173
Lem, Stanislaw 26

liiku 173, 174, 192, 193, 195
limerick 263
lineaarinen tehokkuus **310**
Linné, Carl von 142
lisaanny 192
lisaaViiteLaskuria 50, 424
`<list>` 317
`list` 315
lista (STL) **315**
lisäys-iteraattori **336**
lisääjä *katso* "lisäys-iteraattori"
logaritminen tehokkuus **310**
lokaalisuusperiaate **119**, 133
Lokiviesti::Lokiviesti 154
loppuiteraattori 333
`lower_bound` 322
lukumääräsuhde 132
LukuPuskuri::katso 316
LukuPuskuri::lisaa 316
LukuPuskuri::lue 316
LukuPuskuri::onkoTyhja 316
Luo 34, 52, 114
luo 43
Luokannimi::Luokannimi 81
luoKayttoliittyma 233
luokka
  -funktio **252–253**
  -invariantti 243, 246
  -muuttuja **250–252**
  -olio **249**
  -vakio 251
  `::` 42
  ajoaikainen kysyminen 165–167
  ali- **144**
  attribuutti 122
  dualismi 58
  ennakkoesittely 112
  esittely 60, **60**, 61
  julkinen rajapinta 101, 102
  kanta- **144**
  kuvaaminen (UML) 128
  käyttö 66–67
  löytäminen 123
  meta- 249
  moduulina 58
  rajapinnan näkyvyys 101
  rajapinta- 177, **190–198**
  rajapintatyyppi 253
  rooli 132
  sisältö 60
  tarjotut palvelut 122
  tietotyyppinä 59
  vastuualue 122
  ystäväfunktio 259
  ystäväluokka 260
luokkahierarkia **144**
luokkainvariantti 246
luokkakaavio 127
luokkamalli **287–290**
  erikoistus 296
  osittaiserikoistus **297**
luoLukko 261
luoPaivaysOlio 50
läpikutsufunktio **180**

maailmankaikkeus 26
main 44, 46, 47, 68, 88, 163, 167, 222, 225, 299, 379, 415, 417, 418
malli **283–301**
  erikoistus **295–297**, 324
  `export` 299
  funktio- **286–287**
  instantiointi **285**
  jäsenfunktio- **289**

koodin sijoittelu 298–299
luokka- **287–290**
malliparametri **294–295**
oletusparametri **292–293**
syntaksi 285
typename 285, 299–301
tyyppi vai arvo 299–301
tyyppiparametri **285**
tyyppiparametrien vaatimukset 290
vakioparametri **294**
mallikieli **268**
malliparametri *katso* “tyyppiparametri”
<map> 321, 322
map 321–322
matalakopiointi **204–205**
max 299, 414
<memory> 384
merge 339
metafunktio **352**
metaluokka **249–250**
metaohjelma **349**
metaohjelmointi **349**
käännösaikainen **350**
metodi 64
min 286, 292, 355, 356, 361
minimitakuu **390**
mitätöidä (iteraattori) **334**
mixin 178
Mjono::kloonaa 214
Mjono::Mjono 207
Modula-3 34, 51
modulaarisuus **118**
moduuli **31–34**
C 39–41
C++ 41–50
elinkaari 37
jako käännösyksiköillä 39
käyttäjä 32
lokaalisuusperiaate 119
riippuvuus 118
toteuttaja 32
vastuualue 122
monijoukko (STL) **319–320**
moniperiytyminen 149, **175–190**, 194
erotteleva **186**
salmiakki- 190
toistuva 186
toistuva kantaluokka 190
virtuaalinen **187**
yhdistävä **187**
monirekisterointi 320
monirekisterointi2 321
moniselitteisyys **181–186**, 187
muistin loppuminen 87, 370
muistivirheet 370
muistivuotojen välttäminen 91–92, *katso* “automaattiosoitin”
mukautuva järjestelmä **351**
multimap 322
multiset 319–320
muna-kana -ongelma **112**, 119
muni 192, 193
muodostin *katso* “rakentaja”
muodostuminen 134
Murtoluku::Murtoluku 234
mutable 108
muunnosjäsenfunktio **236**
muutanKuitenkin 109
myohassako 165

namespace 41
nappulaaPainettu 233
NaytaIlmoitus 278, 279
naytaViesti 73, 76

- naytaViesti1 78
- naytaViesti2 78
- neliöllinen tehokkuus **310**
- new 89
- `<new>` 87, 89
- **`new`** 86–89
- **`new`**`(nothrow)` 89
- **`new[]`** 90
- nimeämiskäytäntö 62
- nimeämätön nimiavaruus **49**
- nimiavaruus **41–50**
  - **`::`** 42
  - alias 46
  - hyödyt 44
  - nimeämätön 49
  - näkyvyystarkenninoperaattori 42
  - **`std`** 45
  - synonyymi 46
  - syntaksi 42
  - **`using`** 47
- nollaaAlkiot 333
- `nothrow` 89
- nothrow-takuu **392**
- näkyvyysmääre **101–104**
  - ja periytyminen 150–152
  - oletus 101
- näkyvyystarkenninoperaattori **44**

- O-notaatio **309**
- odotaOKta 76
- ohjelmistokriisi **27**
- ohjelmistosuunnittelu **117**
- OkDialogi 73, 76
- OkDialogi::˜OkDialogi 78
- OkDialogi::OkDialogi 78
- OKodotus 73
- oletusrakentaja **82–83**
  - kääntäjän luoma 83
- olio
  - alustamaton 218
  - alustaminen *katso* “rakentaja”
  - arvo 216
  - arvoparametrina 224–227
  - dynaaminen luominen ja tuhoaminen 85–86
  - elinkaari 71–72
  - identiteetti 57, 58
  - instantiointi 56
  - kahteen kertaan tuhoaminen 92–94
  - kerrosrakenne 149
  - kommunikointi 36
  - kopiointi **202–215**
  - kuvaaminen (UML) 128
  - käyttö tuhoamisen jälkeen 92–94
  - luokka 56, 57
  - luomistoimenpiteet 70–71
  - omistusvastuu 91
  - paluuarvona 224
  - poikkeus- 374
  - rajapinta 95
  - sijoitus **215–222**
  - simulointi 54
  - tietotyyppi 57
  - tila 37, 55, 202
  - tilan eriyttäminen 405–407
  - tilan vaihtaminen 407–408
  - todellinen vs. ohjelmallinen 55
  - tuhoamistoimenpiteet 71

tuhoutuminen *katso* “purkaja”
vakio- 106–108
väliaikais- 225
olio-ohjelmointi **36**
oliomäärittely **35**
oliosuunnittelu **35**, 116–141
oliot
funktio- **343**, 340–348
Ω-notaatio **309**
omistussuhde 134
omistusvastuu 91
onkoAlle5 342
OnkoAlle::OnkoAlle 344
OnkoAlle::operator 344
onkoKarkauspaivaa 68
onkoTyhja 316
operaatio *katso* “jäsenfunktio”
operator 344
optimointi
`inline` 413
ja mallit 295
ja STL 318
osittava jaottelu **118**
osoitin
-aritmetiikka 332
automaatti- *katso* “automaattiosoitin”
funktio- **341**
ja ennakkoesittely 112
ja iteraattorit 332
jäsenmuuttuja 133
kantaluokka- 155
kelvollisuus 335
`this` 66
vaarat 411
vakio- 109
viite osoittimeen 411
älykäs 387
osoitinvakio 111
`ostream_iterator` 337
otsikkotiedosto **39**, 60
suojaus useaan kertaan käytöltä 40
otsikkotiedostot
ISOC++ 45
pääte 46

package (JAVA) 259
paint 52
PaivattyLokiviesti::PaivattyLokiviesti 154
PaivattyMjono::kloonaa 214
PaivattyMjono::PaivattyMjono 208
Paivays::˜Paivays 84
Paivays::annaPaiva 107, 258
Paivays::Paivays 80
pakka (STL) **315**
pakkaus (JAVA) **259**
paluuarvo ja automaattiosoitin 385
palvelu *katso* “jäsenfunktio”
parametri
arvonvälitys **224**
viite- 412
Parnas’s Principles 96
Parnas, David 96, 121
Parnasin periaatteet 96
`partition` 339
peittäminen **168**
periytetty luokka *katso* “aliluokka”
periytyminen **143, 144**
dynaaminen sitominen **160–164**
erikoistaminen **147**

hinta 169–170
ja kopiorakentaja 207–208
ja laajentaminen 156–159
ja luokkainvariantti 247
ja näkyvyys 150–152
ja purkajat 154–155
ja sijoitusoperaattori 219
ja tyyppimuunnos 164–165
ja vastuu 153, 160
luokan kysyminen 165–167
moni- *katso* “moniperiytyminen”
peittäminen **168**
perusteet 149–150
rajapintaluokka **190–198**
syntaksi (C++) 150
toistuva 186
uudelleenkäyttö 147–148
viipaloituminen 156, **210–215**, 221–222
vs. kooste 198–199
yleistäminen **147**
periytymishierarkia **143–147**
perustakuu **390**
pienin 353
PieniPaivays::annaPaiva 65
PieniPaivays::asetaPaiva 65
PieniPaivays::asetaPaivays 65
PieniPaivays::sijoitaPaivays 105
pimpl 405–407
pino (STL) **325**
poikkeus **367**
-kategoriat **371–373**, 375
-käsittelijä **374**
-määreet **382–383**
-neutraalius **392**
-olio 374
-takuu **389**
-turvallinen sijoitus 399–408
dynaamisesti luodut oliot 380
heittäminen 374
ja olioiden tuhoaminen 380–381
ja purkajat 380, 397
ja rakentajat 393
käyttö 374–377
minimitakuu **390**
muistin loppuminen 87
nothrow-takuu **392**
perustakuu **390**
sieppaamaton 377–378
sieppaaminen 374
sisäkkäiset 378–379
uudelleenheittäminen 379
vahva takuu **391**
valvontalohko **374**
vuotaminen 375
yleiskäsittelijä **378**
poistaLukko 261
poke 415
polymorfismi 146
`pop` 325, 326
`pop_back` 313, **313**, 315, 317
`pop_front` **313**, 315, 317
Pratchett, Terry 142
prioriteettijono (STL) **326**
`priority_queue` 326
`private` 101, 103–104, 151
`protected` 101, 152
`public` 101–103, 151
puhdas virtuaalifunktio **172**
puhelinluettelo 322

PuhLuettelo::lisaa 324
PuhLuettelo::poista 324
PuhLuettelo::tulosta 324
purkaja 76, **83–85**
  ja periytyminen 154–155
  ja poikkeukset 380
  ja virtuaalifunktiot 170–171
  kutsuhetki 84
  kutsumatta jättäminen 86, 246
  virtuaali- **168–169**
push 325, 326
push_back 312, 313, **313**, 315, 317
push_front **313**, 315, 317
PVM::PVM 114
pysyvyys- ja vaihtelevuusanalyysi **264**
pysyväisväittämä *katso* “invariantti”

<queue> 326
queue 326

rajapinta **31**, **95–99**, **240**
  -funktio **31**
  -tyyppi **253–258**
  hyvän rajapinnan tunnusmerkkejä 97–98
  julkinen 32
  jäsenmuuttuja 102
  kuvaaminen (UML) 128
  luettelo erilaisista 98
  luokan ja olion 248
  luokan julkinen 101
  luokkavakio 251
  lupaus 147
  sisäinen 258
  sopimus 33, 240
  suunnittelu 33, 96
  tiedon kätkentä 33
  tietotyypit 253
  toteuttaminen (UML) 137
rajapintaluokka 177, **190–198**
rakentaja 74, 76, **80–83**
  alustuslista **81**
  ja periytyminen 152–154
  ja virtuaalifunktiot 170–171
  kopio- 83, **206–210**, 311
  oletus- **82–83**
  tyyppimuunnoksena 233–236
random_shuffle 339
rbegin 336
reflektio **349**
reinterpret_cast 231–232
rekisterointi 319
rekisterointi_optimoitu 320
rend 336
riippuvuus 131
robustness 120
role 132
rooli
  luokka (UML) 132
  suunnittelumalli 270
roskienkeruu **71**, 72, 74
RTTI 164
Run-Time Type Information 164
rutiini *katso* “jäsenfunktio”

saakoHameenKuukaudessa 111
Sandman 69

sarja (STL) **312**
<set> 318, 320
set 318–319
Shakespeare, William 54
siirry 364
siirry_apu 364
siivoa 76
Siivous 73
siivousfunktio1 381
siivousfunktio2 382, 385
siivousfunktio3 386
siivoustoimenpiteet 71, 378
sijoita 222
sijoituksen estäminen 220
sijoitus **215–222**
- automaattiosoitin 385–387
- itseen 218–219
- poikkeusturvallinen 399–408
- viipaloituminen 221–222

sijoitusoperaattori **216–220**
- estäminen 220
- ja periytyminen 219
- kääntäjän luoma 219–220

silmukkaviittaus 119
silta **274–276**
Simula-67 54
sisäinen rajapinta 258
sisäinen tyyppi *katso* “rajapintatyypit”
sisäkkäiset valvontalohkot 378–379
size **313**, 325, 326
Smalltalk 204–206, 249
- metaluokka 249

sopimus rajapinnasta 241
sopimussuunnittelu **240–245**
sort 339
sovelluskehys **199–200**
sovitin
- iteraattori- **336–337**, 337
- säiliö- 304, **325–326**

spagettikoodi 29
splice 317
staattinen elinkaari 77–79
<stack> 325
stack 325
Standard Template Library katso “STL”
static 40, 49
static_cast 229
std 45
STL 302–348
- algoritmit **337–340**
- assosiaatiomonitaulu **322**
- assosiaatiotaulu **321–322**
- assosiatiivinen säiliö **317–318**
- avain 317
- bittivektori **325**
- funktio-olio **343**, 340–348
- geneerisyys 302
- indeksointi 313, 315, 321
- iteraattori *katso* “iteraattori”
- ja automaattiosoittimet 387
- ja perustaulukot 312
- ja string 312
- ja viitteet 311
- jono **326**
- joukko **318–319**
- kaksipäinen jono *katso* “pakka”
- kertaluokka **309**
- kirjallisuus 304
- lista **315**

monijoukko **319–320**
muistinhallinta 311
optimointi 318
pakka **315**
perusteet 303–305
pino **325**
prioriteettijono **326**
sarja **312**
säiliö **310–311**
säiliösovitin **325–326**
totuusvektori **323–324**
vector<**bool**> 323–324
vektori **312–315**
viipaloituminen 212
väli **329**
STLport 335
string 238
<string> 418
string 416–419
substantiivihaku 123
sulkeuma 343
summaa 295, 300
summaaLuvut 376
suoritusaika 309
suunnittelumalli **267–270**
iteraattori **273–274**
kohdistin 273
kokoelma **271–273**
silta **274–276**
suurten kokonaisuuksien hallinta 29
switch 428
syklinen viittaus 119
synonyymi *katso* “viite”
Systema Naturae 142
syväkopiointi **205–206**
syöttöiteraattori **329**
säiliö 304, **310–311**
-sovitin 304
assosiatiivinen **317–318**
ja iteraattorit 332–334
ja viitteet 311
läpikäyminen 326, 333
sarja **312**
viipaloituminen 212
säiliösovitin **325–326**

taikuri 27
tapahtumasekvenssi **138–139**
taulukko-**delete** 90
taulukko-**new** 90
tehokkuuskategoriat **308–310**
tehokkuusvaatimus 307
template 285
template-metaohjelmointi **350**
terminate 377
Θ-notaatio **309**
this 66
throw 375
tiedon kätkentä **33**, 118
tietojäsen *katso* “jäsenmuuttuja”
tietorakenne **30**
jaotteluperusteet 306–308
kertaluokka **309**
luokka 60, 66
läpikäyminen 326
rajapinnat 307
tehokkuuskategoriat **308–310**
Tila 406, 409
tilakone **139–140**
tilan eriyttäminen 405–407
tilan vaihtaminen 407–408
top 325
toteutusmalli **267**
totuusvektori (STL) **323–324**
trait 353

Trurl 26
try 375
Tuhoa 114
tuhoutuminen
    olion *katso* “purkaja”
Tulosta 34, 52
tulosta 43, 323
TulostaAikaisempi 36
tulostaAlkiot 333
tulostaAlle 346
tulostaAlle2 349
tulostaAlle5 342
tulostaKirjat 163
tulostaKTeostiedot 184, 185
tulostaPalautusaika 230
tulostaTiedot 160, 182, 185
tulostus-iteraattori **330**
tuotaNeliotaulukko 386
tuotaTaulukko 386
tuplaa 412
type_info 166
typedef 332
typeid 166
<typeinfo> 166
typename 285, 299–301
tyyppimuunnos 227–236
    ajoaikainen 231
    C 228
    const_cast 230–231
    dynamic_cast 164–165, 231
    implisiittinen 256, 287, 291
    ja rakentaja 233
    käännösaikana 229
    ohjelmoijan määrittelemä 232
    olio-ohjelmointi 228
    reinterpret_cast 231–232
    static_cast 229
    uusi C++ syntaksi 228
Tyyppinimi 86
tyyppiparametri **285**
    vaatimukset 290

UML
    assosiaatio 132
    interface 128
    kooste 134
    koostuminen 135
    lukumääräsuhde 132
    luokka 128
    luokkakaavio 127
    luokkien väliset yhteydet 130
    muodostuminen 134–135
    olio 128
    periytyminen 137
    rajapinta 128
    rakennekuvaus 140
    riippuvuus 131
    tapahtumasekvenssi 138
    tilakone 139
    toteuttaminen 137
uncaught_exception 398
Unified Modelling Language 126
upper_bound 322
using 47
using namespace 48
uusiTyyppi 228
UusiVirheIkkuna 279

vahva takuu **391**
vakio **105–111**
    -iteraattori 332

-jäsenfunktio 107
-muuttuja 105
-olio 106
-osoitin 109
-viite 109, 224, 412
osoitin- 111
perustietotyyppi 105
suunnitteluvirhe 231
tyyppimuunnos 230
vakioaikainen tehokkuus **310**
valvontalohko **374**
vanilla 178
vappu 51
varain **305**
varmistusehto 246
varmistusrutiini **245**
vastuualue **38**, 122
elinkaari 69
ja periytyminen 153, 160
luokka **122–123**
moduuli 32
moduulin toteuttaja 32
rajapinnan sopimus 240
rajapinta 95–99
`<vector>` 313, 415
`vector` 312–315, 414–416
`vector<bool>` 323–324
vektori (STL) **312–315**
viipaloituminen 156, **210–215**, 221–222
viite **411**, 410–413
-parametri 412
ja ennakkoesittely 112
ja säiliöt 311
kantaluokka- 155
kelvollisuus 335
osoittimeen 411
paluuarvona 413
syntaksi 411
vakio- 109
vakioparametri 412
virheellinen 413
viitekopiointi **203–204**
viitelaskuri 387
virheet
hallittu lopetus 369
jatkaminen 370
peruuttaminen 370, 391
toipuminen 370
virheet ohjelmissa **367–368**
virhehierarkia **371–373**
VirheIkkuna::˜VirheIkkuna 278
VirheIkkuna::NaytaIlmoitus 278
virtaiteraattori **336**
virtuaalifunktio **159–160**
hinta 169–170
puhdas **172**
rakentajissa ja purkajissa 170–171
virtuaalinen moniperiytyminen **187**
virtuaalipurkaja **168–169**
virtuaalitaulu 170
virtuaalitauluosoitin 170
`virtual` 159, 168
unohtaminen 168, 169
vuosi 2000 -ongelma **31**, 33, 256
väli (STL) **329**
väliaikaisolio 225

yhdistävä moniperiytyminen **187**
yhtasuuruus 359
yleiskäyttöisyys 264

yleispoikkeuskäsittelijä **378**
yleistäminen **147**
yli-indeksointi 313
yliluokka *katso* “kantaluokka”
ys::tulosta 42
ystäväfunktio **259–260**
ystäväluokka **260–261**

zombie-olio 94

älykäs osoitin 387