18

Moni- ja virtuaaliperiytyminen

Vallitseva periytymismalli käytännön C++-sovelluksissa on julkinen periytyminen yhdestä kantaluokasta. Voimme yleensä olettaa, että useimmissa periytymisen käyttötilanteissa päädymme tähän samaan malliin. Joissakin tilanteissa yksiperiytyminen on kuitenkin riittämätön ratkaisu, koska se joko (1) epäonnistuu mallintamaan ohjelman pääabstraktiota tai (2) malli, jonka se tarjoaa, on tarpeettoman monimutkainen ja vaikeatajuinen. Näissä tapauksissa moniperiytyminen tai sen erikoistaminen — virtuaaliperiytyminen — on parempi ratkaisu. Tässä luvussa keskitytään pääosin C++:n tukeen moni- ja virtuaaliperiytymiselle.

18.1 “Näyttämön” järjestely

Ennen kuin katsomme tarkemmin moni- ja virtuaaliperiytymistä, perustelemme lyhyesti niiden käyttöä. Ensimmäinen esimerkkimme on otettu 3D-tietokonegrafiikasta. Ennen kuin voimme esitellä ongelman, pitää kuitenkin esitellä ongelma-alue.

Näkymä esitetään tietokoneessa *näkymäkäyränä*. Näkymäkäyrä sisältää jotain geometriaa (3D-mallit), yhden tai useamman valon (ilman valoa mallit verhoutuvat pimeyteen), kameran (ilman kameraa emme näe näkymää) ja useita muunnossolmuja, joilla elementit positioidaan.

Renderointi on prosessi, jossa käytetään valoa ja kameratietoa geometriaan, jolla saadaan aikaan 2D-kuva näytölle. Renderointiin liittyvät kaksi pääasiaa ovat (1) valolähteiden luonne, jotka valaisevat näkymää ja (2) geometristen pintamateriaalien piirteet kuten väri, läpinäkyvyys ja läpikuultavuus. Esimerkiksi keijukaisen kuunvalkeiden siipien höyhenet valaistaan melko eri tavalla kuin timanttikuvioiset kyyneleet, jotka vierivät sen silmistä, vaikka ne molemmat valaistaan samalla kultahopeisella valolla.

Sekä valaistuksen että geometrian ominaisuuksien lisääminen, uudelleenasettelu ja kääntely jokaiseen näkymään on eräs tietokonetaiteilijan työläimmistä tehtävistä. Meidän tehtävämme on tehdä vuorovaikutteinen tuki näkymäkäyrän näytöllä tapahtuvaan käsittelyyn. Kuvitellaan, että olemme valinneet nykyiseksi työkaluksemme Open Inventor C++ -rungon (katso [WERNECKE94]), jolla toteutamme taustalla olevan näkymäkäyrän ja jota laajen-

namme alityypityksellä saadaksemme omat tarpeelliset luokka-abstraktiot. Open Inventor tukee esimerkiksi seuraavia kolmea sisäistä valolähdettä, jotka on johdettu abstraktista SoLight-kantaluokasta:

```
class SoSpotLight : public SoLight { ... };
class SoPointLight : public SoLight { ... };
class SoDirectionalLight : public SoLight { ... };
```

jossa So on etuliite, jolla saadaan yksilölliset nimet muutoin yleisistä grafiikka-alan nimistä (runko suunniteltiin ennen nimiavaruuksien esittelyä). Pistevalo (*point light*) on valolähde, joka säteilee kaikkiin suuntiin (ajattelepa aurinkoa). Suuntavalo (*directional light*) säteilee yhteen tiettyyn suuntaan. Kohdevalo on kartiomainen valo kuten teatterituotannoissa käytetty valo, jolla valaistaan tietty näyttämön osa.

Oletusarvo on, että Open Inventor renderoi näkymäkäyrän näytölle käyttäen OpenGL:ää (katso [NEIDER93]). Vaikka se riittää vuorovaikutteiselle näytölle, melkein kaikki kuvat, jotka generoidaan elokuvateollisuuden käyttöön, renderoidaan Pixarin RenderMan:illa (katso [UPSTILL90]). Osatehtävänämme on lisätä omat erikoistetut valotyypit, jotta voisimme tukea näkymäkäyrän renderointia RenderMan:illa:

```
class RiSpotLight : public SoSpotLight { ... };
class RiPointLight : public SoPointLight { ... };
class RiDirectionalLight : public SoDirectionalLight { ... };
```

Tämä toimii odotetulla tavalla. Uudet alityyppimme sisältävät lisätiedon, joka on välttämätöntä RenderMan:illa renderointiin. Open Inventor -kantaluokka silti mahdollistaa renderoinnin samanaikaisesti OpenGL:n kautta. Asiat menevät pieleen, kun tukeamme pitää laajentaa varjoihin.

RenderMan:issa kohde- ja suuntavalo tukevat varjojen luomista (kutsumme näitä *varjostuskykyisiksi valolähteiksi* (*shadow-capable light sources* [SCLS])); pistevalo ei tue. Yleinen algoritmi vaatii, että käymme läpi kaikki näkymän valolähteet ja generoimme *varjokartan* jokaiselle SCLS:lle, joka on päällä. Ongelma on, että valot on tallennettu näkymäkäyrään monimuotoisina SoLight-olioina. Vaikka voimme kapseloida yhteisen tiedon ja tarpeelliset operaatiot SCLS-luokkaan, ei ole selvää, kuinka luokka siepataan olemassaolevaan Open Inventor -hierarkiaan.

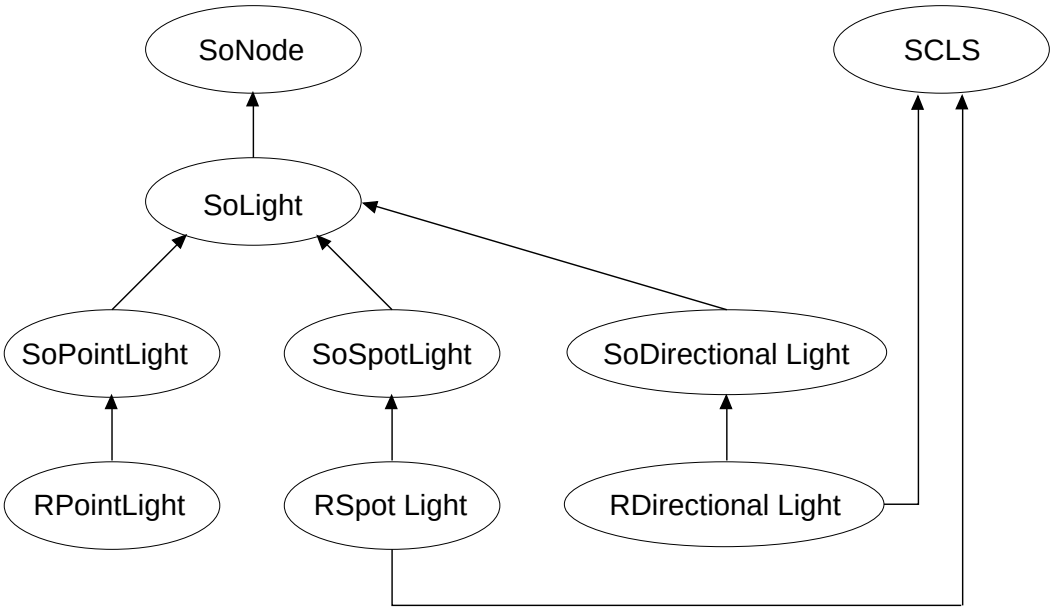
Open Inventor:in SoLight-alipuussa ei ole olemassa käyttökelpoista paikkaa, johon voitaisiin johtaa SCLS-luokka yksin niin, että sekä SdRiSpotLight että SdRiDirectionalLight voidaan myöhemmin johtaa siitä. Ilman moniperiytymistä, parasta, mitä voimme tehdä, on yhdistää kumpikin SCLS-valokategoria SCLS-jäsenluokkaolion kanssa ja tehdä metodi sopivan operaation käynnistämiseen:

```
SoLight *plight = next_scene_light();

if ( RiDirectionalLight *pdilite =
    dynamic_cast<RiDirectionalLight*>( plight ))
        pdilite->scls.cast_shadow_map();
else
if ( RiSpotLight *pslite =
    dynamic_cast<RiSpotLight*>( plight ))
        pslite->scls.cast_shadow_map();
// jne ...
```

(dynamic_cast-operaattori on osa suorituksenaikaista tyyppitunnistusmekanismia (Run-Time Type Identification (RTTI)). Se tukee suorituksenaikaista kyselyä monimuotoisen osoittimen tai viittauksen osoittaman olion todellisesta tyypistä [RTTI käsitellään luvussa 19].)

Moniperiytymisessä kapseloimme SCLS-alityypit ja siten suojaamme koodimme siltä, että sitä muokattaisiin jokaisen SCLS-kategorian valolähteen lisäyksen ja poiston yhteydessä. Tämä tulee esille kuvassa 18.1.



Kuva 18.1 Valohierarkian moniperiytyminen

```
class RiDirectionalLight :
    public SoDirectionalLight, public SCLS { ... };

class RiSpotLight :
    public SoSpotLight, public SCLS { ... };

// ...
```

```
SoLight *plight = next_scene_light();
if ( SCLS *pscls = dynamic_cast<SCLS*>(plight))
   pscls->cast_shadow_map();
```

Tämä on silti hieman epätäydellinen ratkaisu. Jos olisimme päässeet Open Inventor:in lähdekoodiin, olisimme tulleet toimeen moniperiytymisessä lisäämällä SCLS-osoitinjäsenen SoLight-luokkaan, joka olisi lisännyt tukea cast_shadow_map()-operaatioon:

```
class SoLight : public SoNode {
public:
     void cast_shadow_map()
          { if ( _scls ) _scls->cast_shadow_map(); }
     // ...
protected:
     SCLS *_scls;
};

// ...

SdSoLight *plight = next_scene_light();
plight->cast_shadow_map();
```

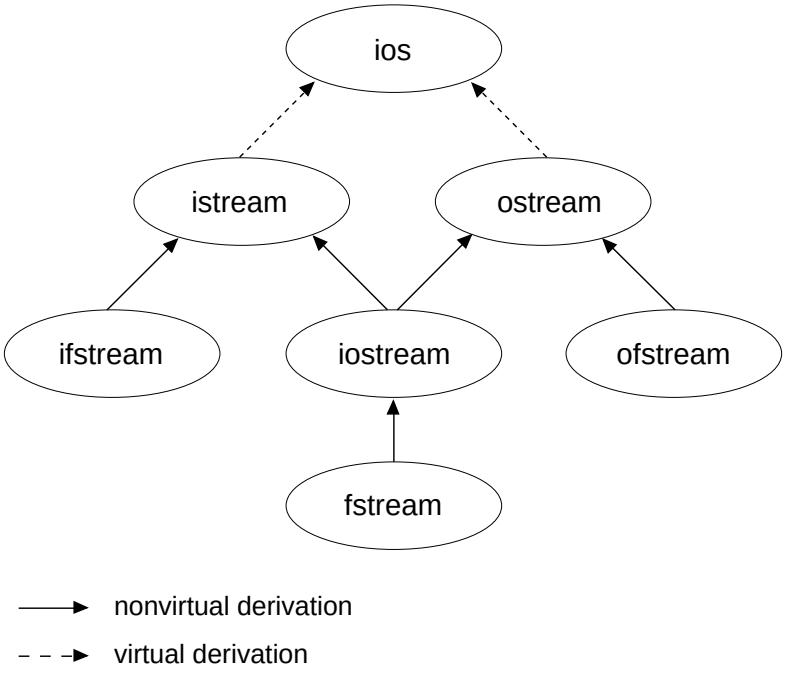
Kaikkein laajimmin käytetty käytännön sovellutus moniperiytymisestä (ja virtuaaliperiytymisestä) on C++-vakiokirjaston syötön ja tulostuksen iostream-kirjasto. Käyttäjän nähtävissä olevat kaksi iostream-pääluokkaa ovat istream-luokka (syöttöön) ja ostream-luokka (tulostukseen). Molemmille luokille yhteisiin piirteisiin kuuluvat

1. Muototilatieto (on kokonaisarvo desimaalisena, oktaalisena tai heksadesimaalisena ilmaisuna; on liukulukuarvo kiinteänä desimaalisena tai tieteellisenä ilmaisuna jne.)
2. Ehtotilatieto (onko stream-olio hyvässä vai huonossa tilassa jne.)
3. Paikallistieto (tuleeko päivä vai kuukausi ensin näytölle, kuten 7/4/76 jne.)
4. Varsinainen puskuri, joka sisältää luettavan tai kirjoitettavan tiedon

Nämä yhteiset piirteet on laitettu abstraktiin ios-kantaluokkaan, josta sekä istream- että ostream-luokka on johdettu.

Iostream-luokka on toinen esimerkki moniperiytymisestä. Se tukee saman tiedoston sekä lukemista että kirjoittamista. Se periytyy sekä istream- että ostream-luokista. Valitettavasti oletusarvo on, että se perii myös kaksi erillistä ilmentymää ios-kantaluokasta, joita emme joko tarvitse tai pysty käsittelemään helposti.

Virtuaaliperiytyminen on ratkaisu ongelmaan, kun peritään useita



Kuva 18.2 Iostream-hierarkian virtuaaliperiytyminen (yksinkertaistettu)

kantaluokan ilmentymiä, kun tarvitaan vain yksi, jaettu ilmentymä. Kuvassa 18.2 on yksinkertaistettu kuva iostream-luokkahierarkiasta.

Jaeltavaksi tarkoitettujen olioiden käsittelyn tuki (Distributed Object Computing) on toinen käytännön esimerkki virtuaali- ja moniperiytymisestä. Jos haluat nähdä tarkemmin siitä keskusteluja ja kuvauksia, katso artikkelit, jotka ovat toimittaneet Douglas Schmidt ja Steve Vinoski julkaisussa [LIPPMAN96b].

Tämän luvun pääaiheita ovat moni- ja virtuaaliperiytyminen. Keskitymme tässä niiden käyttöön ja käyttäytymiseen. Oheislukemistossamme, *Inside the C++ Object Model*, käsitellään edistyneempiä suorituskyky- ja suunnitteluaiheita.

Seuraavassa käsittelyssä olemme valinneet pedagogisemman (opettavaisemman) esimerkin — tarkoittaa eläintarhan eläinten hierarkiaa. Eläintarhamme eläimet esiintyvät abstraktion eri tasoilla. On tietysti yksilöllisiä eläimiä kuten Ling-ling, Mowgli ja Balou. Jokainen kuuluu eläinlajiin; Ling-ling esimerkiksi on jättiläispanda. Lajit ovat vuorostaan heimojen jäseniä. Jättiläispanda on Bear-heimon (karhuheimon) jäsen, vaikka kuten näemme kohdassa 18.5, tuo suhde oli pitkään kiistelyn aiheena eläintieteeseen erikoistuneiden keskuudessa. Jokainen heimo vuorostaan on eläinkunnan jäsen — tässä tapauksessa rajoitetumman kunnan eli tietyn eläintarhan.

Jokainen abstraktiotaso sisältää tietoa ja operaatioita, jotka tukevat laajempaa käyttäjäkategoriaa. Esimerkiksi abstrakti ZooAnimal-luokka sisältää tietoa, joka on yhteistä kaikille eläintarhan eläimille ja toimii julkisena rajapintana kaikille yleisille kyselyille, joita voidaan tehdä. Bear-luokka sisältää tietoa, joka on yksilöllistä karhuheimolle jne.

Varsinaisten eläintarhan eläimien lisäksi on apuluokkia, joihin on kapseloituna useita abstraktioita kuten uhanalaiset eläimet. Esimerkiksi Panda-luokan toteutuksessamme Panda on monijohdettu Bear- ja Endangered-luokista.

18.2 Moniperiytyminen

Jotta moniperiytymistä voitaisiin tukea, johdettujen luokkien luetteloa kuten

```
class Bear : public ZooAnimal { ... };
```

laajennetaan niin, että se tukee pilkuin eroteltua kantaluokkien luetteloa. Esimerkiksi:

```
class Panda : public Bear, public Endangered { ... };
```

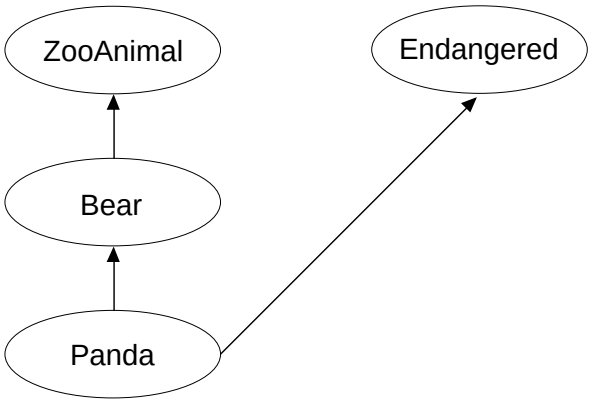
Jokaisen luetellun kantaluokan pitää myös määrittää sen käsittelytaso eli joko `public`, `protected` tai `private`. Kuten yksiperiytymisen yhteydessä, voidaan kantaluokka moniperiytymisessä luetella vain, jos sen määrittely on jo nähty.

Ei ole olemassa kieleen liittyvää rajoitusta niiden kantaluokkien lukumäärälle, joista luokka voidaan johtaa. Käytännössä kaikkein yleisin näyttää olevan kaksi kantaluokkaa, joista toinen usein edustaa julkista, abstraktia rajapintaa ja toinen muodostaa yksityisen toteutuksen (vaikka kumpikaan kahdesta aikaisemmasta esimerkistä ei kuvaa tätä tapausta). Johdetut luokat, jotka periytyvät kolmesta tai useammasta lähimmästä kantaluokasta, noudattavat *yhdistelypohjaista* suunnittelutyyliä, jossa jokainen kantaluokka edustaa yhtä *tahoa* johdetun valmiista rajapinnasta.

Moniperiytymisen yhteydessä johdettu luokka sisältää kantaluokan aliolion jokaisesta sen kantaluokasta (katso kohdasta 17.3, jossa käsitellään johdetun luokan kantaluokan alioliota). Kun esimerkiksi kirjoitamme

```
Panda yin_yang;
```

`yin_yang` muodostuu Bear-luokan alioliosta (joka itse sisältää ZooAnimal-kantaluokan aliolion), Endangered-luokan alioliosta ja ei-staattisesta tietojäsenistä, jotka on esitelty Panda-luokkaan (katso kuva 18.3), jos niitä on.



Kuva 18.3 Panda-hierarkian moniperiytyminen

Kantaluokan muodostajat käynnistetään johdettujen luokkien luettelon mukaisessa esittelyjärjestyksessä. Esimerkiksi yin_yang:in kohdalla muodostajien käynnistysjärjestys on seuraava: Bear-muodostaja (koska Bear on johdettu ZooAnimal-luokasta; kuitenkin ennen Bear-muodostajan suoritusta käynnistetään ZooAnimal-muodostaja), Endangered-muodostaja, sitten Panda-muodostaja.

Kuten käsittelimme kohdassa 17.4, muodostajien käynnistysjärjestykseen *ei vaikuta* kantaluokan mukanaolo jäsenen alustusluettelossa eikä järjestys, jossa ne on lueteltu. Tämä tarkoittaa, että vaikka Bear:in oletusmuodostaja pitäisi käynnistää automaattisesti eikä sitä siksi mainita jäsenen alustusluettelossa, kuten seuraavassa

```
// Bear:in oletusmuodostaja käynnistetään ennen
// Endangered:in kaksiargumenttista muodostajaa ...

Panda::Panda()
      : Endangered( Endangered::environment,
                    Endangered::critical )
{ ... }
```

käynnistetään Bear:in oletusmuodostaja silti ennen eksplisiittisesti lueteltua, kaksiargumenttista Endangered-muodostajaa.

Samalla tavalla tuhoajien käynnistysjärjestys on aina päinvastainen kuin muodostajien järjestys. Esimerkissämme tuhoajien käynnistysjärjestys on seuraava: ~Panda(), ~Endangered(), ~Bear(), ~ZooAnimal().

Kuten näimme kohdassa 17.3, kantaluokan julkisia ja suojattuja jäseniä voidaan käsitellä aivan kuin ne olisivat johdetun luokan jäseniä. Sama pätee moniperiytymiseen. Moniperiytymisessä on kuitenkin mahdollisuus periä samanniminen jäsen kahdesta tai useammasta kantaluokasta. Tässä tapauksessa suora käsittely on moniselitteistä ja johtaa käännösvirheeseen.

Käännösvirhettä *ei* kuitenkaan laukaista kahden jäsenen tarkentamattoman käsittelyn potentiaalisen moniselitteisyyden takia, vaan ainoastaan siinä tapauksessa, että kun sitä todellisuudessa yritetään (katso käsittely aiheesta kohdasta 17.4.4). Jos esimerkiksi sekä Bear että Endangered määrittelevät print()-jäsenfunktion, silloin lause kuten seuraavassa

```
yin_yang.print( cout );
```

johtaa käännösvirheeseen, vaikka kahdessa perityssä jäsenfunktiossa määritellään erilaiset parametrityypit:

```
Error:  yin_yang.print( cout ) -- ambiguous, one of
                Bear::print( ostream& )
                Endangered::print( ostream&, int )
```

Syy tähän on se, että perityt jäsenfunktiot eivät muodosta ylikuormitettuja funktioita perittyyn luokkaan (katso käsittely kohdasta 17.3). Tästä syystä print() ratkaistaan käyttäen vain nimiresoluutiota print-nimelle sen sijaan, että käytettäisiin ylikuormituksen ratkaisua, joka perustuu todellisiin argumenttityyppeihin, jotka välitetään print()-funktiolle. (Katsomme kohdassa 18.4, kuinka tämä voitaisiin ratkaista.)

Jos on tarpeen, yksiperiytymisessä johdetun luokan osoitin, viittaus tai olio konvertoidaan automaattisesti julkisesti johdetun kantaluokan osoittimeksi, viittaukseksi tai olioksi. Jälleen: sama pätee moniperiytymiseen. Esimerkiksi Panda-osoitin, viittaus tai olio voidaan konvertoida ZooAnimal-, Bear- tai Endangered-luokan osoittimeksi, viittaukseksi tai olioksi. Esimerkiksi:

```
extern void display( const Bear& );
extern void highlight( const Endangered& );

Panda yin_yang;

display( yin_yang );   // ok
highlight( yin_yang ); // ok

extern ostream&
    operator<<( ostream&, const ZooAnimal& );

cout << yin_yang << endl; // ok
```

Kuitenkin jälleen kerran: moniperiytymisessä on moniselitteisen konversion mahdollisuus paljon suurempi. Mietipä esimerkiksi seuraavia kahta funktiota:

```
extern void display( const Bear& );
extern void display( const Endangered& );
```

Tarkentamaton display()-funktion käynnistys Panda-oliolla kuten

```
Panda yin_yang;
display( yin_yang ); // virhe: moniselitteinen
```

johtaa seuraavaan yleisen käännösvirheen muotoon:

```
Error:  display( yin_yang ) -- ambiguous, one of
                extern void display( const Bear& );
                extern void display( const Endangered& );
```

Kääntäjällä ei ole keinoa erottaa lähimpien kantaluokkien välisiä johdettujen luokkien konversioiden ehtoja. Jokainen konversio on yhtä sopiva. (Katsomme kohdassa 18.4.1, kuinka tämä voitaisiin ratkaista.)

Jotta voisimme nähdä, kuinka moniperiytyminen vaikuttaa virtuaalifunktiomekanismiin, määritelkäämme virtuaalifunktiojoukko jokaiselle Panda:n lähikantaluokalle. (Virtuaalifunktiot esiteltiin kohdassa 17.2 ja käsiteltiin tarkemmin kohdassa 17.5.)

```
class Bear : public ZooAnimal {
public:
        virtual ~Bear();
        virtual ostream& print( ostream& ) const;
        virtual string isA() const;
        // ...
};

class Endangered {
public:
        virtual ~Endangered();
        virtual ostream& print( ostream& ) const;
        virtual void highlight() const;
        // ...
};
```

Määritellään Panda nyt niin, että se saa oman print()-ilmentymän ja muodostajan ja esittelee uuden virtuaalifunktion, cuddle(), kuten seuraavassa:

```
class Panda : public Bear, public Endangered
{
public:
        virtual ~Panda();
        virtual ostream& print( ostream& ) const;
        virtual void cuddle();
        // ...
};
```

Virtuaalifunktiot, jotka voidaan käynnistää suoraan Panda-oliosta, ovat seuraavat:

Taulukko 18.1 Pandan aktiiviset virtuaalifunktiot

Virtaalifunktion nimi	Aktiivinen ilmentymä
destructor	Panda::~Panda()
print(ostream&) const	Panda::print(ostream&)
isA() const	Bear::isA()
highlight() const	Endangered::highlight()
cuddle()	Panda::cuddle()

Kun Bear- tai ZooAnimal-luokan osoitin tai viittaus alustetaan tai siihen sijoitetaan Panda-luokan olion osoite, eivät Panda-kohtaiset ja Endangered-osat Panda-rajapinnasta enää ole käsiteltävissä. Esimerkiksi:

```
Bear *pb = new Panda;

pb->print( cout ); // ok: Panda::print(ostream&)
pb->isA();         // ok: Bear::isA()
pb->cuddle();      // virhe: ei ole osa Bear-rajapintaa
pb->highlight();   // virhe: ei ole osa Bear-rajapintaa
delete pb;         // ok: Panda::~Panda()
```

(Huomaa, että jos Panda-olio oli sijoitettu ZooAnimal-osoittimeen, aikaisemmin kuvatut käynnistykset ratkaistaan samalla tavalla.)

Samalla tavalla, kun Endangered-osoitin tai -viittaus alustetaan tai siihen sijoitetaan Panda-luokkaolion osoite, eivät Panda-kohtaiset ja Bear-osat Panda-rajapinnasta ole enää käsiteltävissä. Esimerkiksi:

```
Endangered *pe = new Panda;

pe->print( cout );// ok: Panda::print(ostream&)

// virhe: ei ole osa Endangered-rajapintaa
pe->isA();

// virhe: ei ole osa Endangered-rajapintaa
pe->cuddle();

pe->highlight(); // ok: Endangered::highlight()
delete pe;       // ok: Panda::~Panda()
```

Virtuaalituhoajan käsittely on vastaava huolimatta osoitintyypistä, jonka kautta olion tuhoamme. Esimerkiksi tässä

```
// ZooAnimal *pz = new Panda;
delete pz;

// Bear *pb = new Panda;
delete pb;
```

```
// Panda *pp = new Panda;
delete pp;

// Endangered *pe = new Panda;
delete pe;
```

tapahtuu täsmälleen samanlainen tuhoajien käynnistysjärjestys. Tuhoajien käynnistysjärjestys on päinvastainen kuin muodostajien järjestys: Panda-tuhoaja käynnistetään virtuaalimekanismin kautta. Seuraavat Panda-tuhoajan, Endangered-, Bear- ja sitten ZooAnimal-tuhoajien suoritukset käynnistetään staattisesti vuorollaan.

Monijohdetun luokan jäsenittäinen alustus ja sijoittaminen käyttäytyy samalla tavalla kuin yksiperiytymisessä (katso käsittely kohdasta 17.6). Esimerkiksi Panda-esittelyn

```
class Panda : public Bear, public Endangered
{ ... };
```

seuraava ling_ling:in jäsenittäinen alustaminen

```
Panda yin_yang;
Panda ling_ling = yin_yang;
```

käynnistää Bear-kopiointimuodostajan (koska kuitenkin Bear on johdettu ZooAnimal-luokasta, käynnistetään ennen Bear-kopiointimuodostajaa ZooAnimal-kopiointimuodostaja), sitten Endangered-kopiointimuodostaja ennen Panda-kopiointimuodostajan rungon käynnistystä. Jäsenittäinen sijoittaminen käyttäytyy samalla tavalla.

Harjoitus 18.1

Mitkä seuraavista esittelyistä ovat virheellisiä, vai onko yksikään? Perustele, miksi.

```
(a) class CADVehicle : public CAD, Vehicle { ... };
(b) class DoublyLinkedList:
        public List, public List { ... };
(c) class iostream:
        private istream, private ostream { ... };
```

Harjoitus 18.2

Kun seuraavassa luokkahierarkiassa jokainen luokka määrittelee oletusmuodostajan

```
class A { ... };
class B : public A { ... };
class C : public B { ... };
class X { ... };
class Y { ... };
class Z : public X, public Y { ... };
class MI : public C, public Z { ... };
```

mikä on muodostajien käynnistysjärjestys seuraavassa määrittelyssä?

```
MI mi;
```

Harjoitus 18.3

Kun on seuraava luokkahierarkia, jossa jokainen luokka määrittelee oletusmuodostajan

```
class X { ... };
class A { ... };
class B : public A { ... };
class C : private B { ... };
class D : public X, public C { ... };
```

mitkä seuraavista konversioista eivät ole sallittuja, vai onko yksikään?

```
D *pd = new D;

(a) X *px = pd; (c) B *pb = pd;
(b) A *pa = pd; (d) C *pc = pd;
```

Harjoitus 18.4

Kun on seuraava luokkahierarkia, jossa on kokoelma virtuaalifunktioita

```
class Base {
public:
    virtual ~Base();
    virtual ostream& print();
    virtual void debug();
    virtual void readOn();
    virtual void writeOn();
    // ...
};

class Derived1 : virtual public Base {
public:
    virtual ~Derived1();
    virtual void writeOn();
    // ...
};

class Derived2 : virtual public Base {
public:
    virtual ~Derived2();
    virtual void readOn();
    // ...
};

class MI : public Derived1, public Derived2 {
public:
    virtual ~MI();
```

```
    virtual ostream& print();
    virtual void debug();
    // ...
};
```

mikä ilmentymä mistäkin funktiosta käynnistetään seuraavassa?

```
Base *pb = new MI;

(a) pb->print();  (c) pb->readOn();  (e) pb->log();
(b) pb->debug();  (d) pb->writeOn(); (f) delete pb;
```

Harjoitus 18.5

Käytä harjoituksessa 18.4 määriteltyä luokkahierarkiaa ja yksilöi aktiiviset virtuaalifunktiot, kun ne käynnistetään seuraavien kautta: (a) pd1 ja (b) d2:

```
(a) Derived1 *pd1 = new MI;
(b) MI obj;
   Derived2 d2 = obj;
```

18.3 Julkinen, yksityinen ja suojattu periytyminen

Julkista johtamista sanotaan *tyyppiperiytymiseksi*. Johdettu luokka on kantaluokan alityyppi; se kumoaa kantaluokan kaikkien tyyppikohtaisten jäsenfunktioiden toteutukset, kun taas perii ne, jotka ovat jaettuja. Johdettu luokka kuvastaa yleensä *is-a*-suhdetta, jossa sillä on erikoispiirteitä vielä yleisemmästä kantaluokastaan. Bear on eräänlainen ZooAnimal. AudioBook on eräänlainen LibBook; molemmat ovat eräänlaisia LibraryLendingMaterial-olioita. Sanomme, että Bear on ZooAnimal:in alityyppi kuten Panda. Samalla tavalla sanomme, että AudioBook on LibBook:in alityyppi ja että molemmat ovat LibraryLendingMaterial:in alityyppejä. Alityyppi voidaan korvata läpinäkyvästi kaikkialla siellä, missä ohjelma edellyttää sen julkista kantatyyppiä, ja ohjelma jatkaa suoritusta oikein (tietysti edellyttäen, että alityyppi on toteutettu oikein). Kaikki periytymisesimerkkimme tähän saakka ovat heijastaneet alityypin periytymistä.

Yksityistä johtamista sanotaan *toteutusperiytymiseksi*. Johdettu luokka ei tue kantaluokan julkista rajapintaa suoraan, vaan se haluaa käyttää uudelleen kantaluokan toteutusta tekemällä oman julkisen rajapinnan. Kuvataksemme näitä käsitteitä toteutamme PeekbackStack-pinon.

PeekbackStack tukee pinosta hakua peekback()-metodilla

```
bool
PeekbackStack::
peekback( int index, type &value ) { ... }
```

jossa value sisältää elementin indeksissä index, jos peekback() palauttaa totuusarvon tosi. Jos peekback() palauttaa arvon epätosi, index on kelvoton ja value:n arvoksi asetetaan pinon ylimmän elementin arvo.

Tässä on kaksi todennäköistä virhealuetta PeekbackStack-toteutuksessamme:

1. Abstrakti PeekbackStack-toteutus: olemmeko toteuttaneet oikein sen käyttäytymisen?
2. Taustalla olevan esitystavan toteutus: olemmeko hoitaneet oikein muistin varaamisen ja vapauttamisen, pino-olioiden kopioimisen jne.?

Pino toteutetaan yleensä joko taulukkona tai elementtien linkitettynä listana (vakiokirjaston pino muodostuu oletusarvoisesti pakasta (*deque*), vaikka voimme määrittää halutessamme vektorin (*vector*) — katso luvusta 6). Se, mistä pitäisimme, olisi täysin taattu (no, vähintään testattu ja täysin tuettu) toteutus joko taulukosta tai listasta, jonka voisimme vain kytkeä PeekbackStack-pinoomme. Jos voisimme tehdä sen, silloin olisimme vapaita keskittymään siihen, että saamme pinon toimimaan oikein.

Sattumalta meillä on IntArray-luokkamme, jonka toteutimme kohdassa 2.3 (kyllä, käsittelyaiheemme vuoksi jätämme huomiotta sekä vakiokirjaston pakkaluokan että tukemme elementtityyppeille, jotka ovat muun kuin int-tyyppisiä). Kysymys kuuluu sitten, kuinka IntArray-luokkaa voitaisiin parhaiten käyttää hyväksi PeekbackStack-toteutuksessamme. Ensimmäinen ajatuksemme on tietysti periytyminen. (Huomaa, että IntArray-luokkaa ei tarvitse muokata ja muuttaa sen jäseniä yksityisistä julkisiksi.) Tässä on toteutuksemme:

```
#include <IntArray.h>

class PeekbackStack : public IntArray {
private:
   const int static bos = -1;

public:
   explicit PeekbackStack( int size )
        : IntArray( size ), _top( bos ){}

   bool empty() const { return _top == bos; }
   bool full()  const { return _top == size()-1; }
   int  top()   const { return _top; }

   int pop() {
      if ( empty() )
          /* käsittele virhetilanne */ ;
      return ia[ _top-- ];
   }

   void push( int value ) {
       if ( full() )
           /* käsittele virhetilanne */ ;
       ia[ ++_top ] = value;
   }

   bool peekback( int index, int &value ) const;

private:
```

```
    int _top;
};

inline bool
PeekbackStack::
peekback( int index, int &value ) const
{
  if ( empty() )
      /* käsittele virhetilanne */ ;

  if ( index < 0 || index > _top )
  {
      value = ia[ _top ];
      return false;
  }

  value = ia[ index ];
  return true;
  }
```

Tässä saavutetaan täsmälleen se, mitä halusimme — ja enemmänkin. Ohjelmakoodi, joka käyttää uutta PeekbackStack-luokkaamme, saattaa myös käyttää sopimattomasti sen IntArray-kantaluokan julkista rajapintaa. Esimerkiksi:

```
extern void swap( IntArray&, int, int );
PeekbackStack is( 1024 );

// hups: odottamaton PeekbackStack-pinon käyttötilanne
swap(is,i,j);
is.sort();
is[0] = is[512];
```

PeekbackStack-luokan abstraktion tulisi taata “viimeisenä sisään, ensimmäisenä ulos” -käyttäytymistä sisältämiensä elementtien käsittelyssä. Kuitenkin IntArray-rajapinnan lisäkäytettävyys saattaa huonoon valoon sen kaltaisen käyttäytymisen takuun.

Ongelma on, että julkinen johtaminen määrittelee *is-a*-suhteen. PeekbackStack ei kuitenkaan ole eräänlainen IntArray. Sen sijaan PeekbackStack:illä on IntArray:stä osa sen toteutusta. IntArray:n julkinen rajapinta ei ole osa PeekbackStack-luokan julkista rajapintaa. PeekbackStack-luokka toivoo käyttävänsä uudelleen IntArray-luokan toteutusta; PeekbackStack ei kuitenkaan ole IntArray-alityyppi.

Yksityinen kantaluokka heijastaa periytymismuotoa, joka ei perustu alityyppisuhteisiin. Kantaluokan koko julkisesta rajapinnasta tulee johdetun luokan yksityinen. Kumpikaan edellisistä PeekbackStack-luokkailmentymien virheellisistä käyttötilanteista ei ole enää sallittu paitsi johdetun luokan ystäville ja jäsenfunktioille.

Ainoa muutos, joka vaaditaan aikaisempaan PeekbackStack-määrittelyyn, on public-avainsanan korvaaminen private-avainsanalla johdettujen luokkien luettelossa. Muita public- ja private-avainsanoja itse luokkamäärittelyssä ei tarvitse muuttaa.

```
class PeekbackStack : private IntArray { ... };
```

18.3.1 Periytyminen vastaan kooste

PeekbackStack-luokan toteutus yksityisesti johdetusta IntArray-luokasta toimii — mutta onko se välttämätöntä? Onko jotain saavutettu tässä periytymistapauksessa? Tässä tapauksessa ei.

Julkinen periytyminen on tehokas tuki *is-a*-alityyppi-suhteeseen. PeekbackStack-toteutus kuitenkin edustaa *has-a*-suhdetta IntArray-luokan kanssa. PeekbackStack-luokalla on IntArray osana sen toteutusta. *Has-a*-suhdetta tukee yleensä paremmin *kooste* (*composition*) kuin periytyminen. Kooste toteutetaan muodostamalla yhdestä luokasta toisen luokan jäsen. Tässä tapauksessa IntArray-oliosta tehdään PeekbackStack:in jäsen. Tässä on *has-a*-toteutuksemme PeekbackStack-pinomme IntArray-ilmentymästä:

```
class PeekbackStack {
private:
        const int static bos = -1;

public:
        explicit PeekbackStack( int size ) :
        stack( size ), _top( bos ){}

        bool empty() const { return _top == bos; }
        bool full()  const { return _top == stack.size()-1; }
        int  top()   const { return _top; }

        int pop() {
            if ( empty() )
             /* käsittele virhetilanne */ ;
            return stack[ _top-- ];
        }

        void push( int value ) {
            if ( full() )
                /* käsittele virhetilanne */ ;
            stack[ ++_top ] = value;
        }
        bool peekback( int index, int &value ) const;

  private:
      int _top;
      IntArray stack;
  };

  inline bool
```

```
PeekbackStack::
peekback( int index, int &value ) const
{
    if ( empty() )
            /* käsittele virhetilanne */ ;

    if ( index < 0 || index > _top )
    {
            value = stack[ _top ];
            return false;
       }

    value = stack[ index ];
    return true;
}
```

Seuraava on erittäin lavea ohje siitä, pitääkö käyttää koostetta vai yksityistä periytymistä luokkasuunnittelussa, kun *has-a*-suhde on olemassa:

- Jos halutaan kumota kaikki luokan virtuaaliset funktiot, pitää siitä periytyä yksityisesti.
- Jos halutaan mahdollistaa, että luokka viittaa johonkin hierarkian mahdollisista tyypeistä, pitää käyttää viittauskoostetta; käsittelemme tätä tarkemmin kohdassa 18.3.4.
- Jos haluamme yksinkertaisesti, kuten PeekbackStack-luokassamme, käyttää uudelleen hyväksi toteutusta, on kooste parempi kuin periytyminen. Jos halutaan oliolle “laiskaa muistinvarausta”, on viittauskooste (osoitinta käyttäen) yleensä parempi suunnitteluratkaisu.

18.3.2 Yksittäisten jäsenten erivapaudet

IntArray:stä yksityisesti johdetussa PeekbackStack-luokassa kaikki IntArray-luokan suojatut ja julkiset jäsenet peritään yksityisinä jäseninä. Olisi kuitenkin hyvä, jos tämän PeekbackStack-toteutuksen asiakkaat kykenisivät kyselemään PeekbackStack-ilmentymän kokoa:

```
is.size();
```

Luokan suunnittelija voi vapauttaa kantaluokan yksittäisiä jäseniä ei-julkisen johtamisen vaikutuksilta. Esimerkiksi seuraavassa vapautetaan IntArray:n size()-jäsenfunktio:

```
class PeekbackStack : private IntArray {
public:
  // säilytä julkinen saantitaso
  using IntArray::size;
  // ...
};
```

Toinen syy yksittäisten jäsenten vapauttamiseen on sallia myöhempien johdettujen luokkien pääsy yksityisen kantaluokan suojattuihin jäseniin. Oletetaan esimerkiksi, että käyttäjät haluaisivat PeekbackStack-alityypin kasvavan dynaamisesti. Jotta se voitaisiin tehdä, PeekbackStack-luokasta johdetun luokan pitää päästä IntArray:n suojattuihin elementteihin ia ja _size:

```
template <class Type>
class PeekbackStack : private IntArray{
public:
   using IntArray::size;
   // ...

protected:
   using IntArray::_size;
   using IntArray::ia;
   // ...
};
```

Johdettu luokka voi vain tallentaa perityn jäsenen alkuperäiselle käsittelytasolleen. Käsittelytasoa ei voi tehdä enemmän tai vähemmän rajoittavaksi, kuin mitä alkuperäisessä kantaluokassa on määritetty.

Yleinen moniperiytymisen malli on periä luokan julkinen rajapinta ja toisen luokan yksityinen toteutus. Esimerkiksi Booch Components, joka on C++-luokkakirjasto, sisältää kasvavan Queue-jonon, joka on toteutettu seuraavasti (katso Michael Vilotin ja Grady Boochin artikkelit julkaisusta [LIPPMAN96b] saadaksesi lisätietoja):

```
template < class item, class container >
class Unbounded_Queue:
    private Simple_List< item >,// toteutus
    public  Queue< item >       // rajapinta
{ ... };
```

18.3.3 Suojattu periytyminen

Johtamisen kolmas muoto on *suojattu periytyminen*. Suojatussa periytymisessä kaikista kantaluokan julkisista jäsenistä tulee johdetun luokan suojattuja jäseniä. Tämä tarkoittaa, että niitä voidaan käsitellä luokasta myöhemmin johdetuissa luokissa, mutta ei luokkahierarkian ulkopuolelta. Jos esimerkiksi haluaisimme johtaa PeekbackStack:in Stack-luokasta, yksityinen johtaminen

```
// hups: tämä ei tue myöhempää PeekbackStack:in
// johtamista: kaikki IntArray-jäsenet ovat nyt yksityisiä
class Stack : private IntArray { ... };
```

on liian rajoittava, koska IntArray-jäsenten tekeminen yksityisiksi Stack:issa estää myöhempien johdettujen luokkien pääsyn noihin jäseniin. Jotta voitaisiin tukea seuraavaa

```
class PeekbackStack : public Stack { ... };
```

pitää Stack johtaa IntArray:sta suojatusti:

```
class Stack : protected IntArray { ... };
```

18.3.4 Oliokooste

Varsinaisesti on olemassa kaksi oliokoostemuotoa:

1. *Arvokooste*, jossa luokan varsinainen olio esitellään jäseneksi, kuten kuvattiin kaksi alikohtaa sitten uudistetussa PeekbackStack-toteutuksessamme.
2. *Viittauskooste*, jossa oliota osoitetaan epäsuorasti joko viittauksen avulla tai osoittimella luokkaolion jäseneen.

Arvokoosteessa on automaattinen olion elinkaaren ja kopioinnin hallinta, ja sillä on tehokkaampi, suora pääsy itse olioon. Missä tilanteissa viittauskooste on parempi?

Sanokaamme esimerkiksi päättäneemme, että Endangered on parempi esittää koosteena kuin periytymisellä. Pitäisikö meidän määritellä Endangered-olio suoraan ZooAnimal:iin vai viitata siihen epäsuorasti osoittimen tai viittauksen kautta? Mietitäänpä, (1) onko kaikilla ZooAnimal:in eläimillä tämä piirre vai ei, ja (2) muuttuuko piirre ajan mittaan (eli voidaako se joko lisätä tai poistaa).

Jos vastaus kohtaan 1 on, että kaikilla ZooAnimal-olioilla on tämä piirre, silloin arvokooste on yleensä parempi. (*Yleensä* tässä tarkoittaa, että arvokooste ei välttämättä ole kaikkein tehokkain esitystapa suurille luokkaolioille etenkään, jos niitä kopioidaan usein. Tässä tapauksessa viittauskooste mahdollistaa, että voimme välttää tarpeetonta kopioimista, kun sitä käytetään yhdessä strategian kanssa, jota sanotaan *copy on write* (*kirjoita kopio vasta, kun se on välttämätöntä*). (Haittapuoli on olion hallinnan kasvava monimutkaisuus.) Tämän tekniikan käsittely on kuitenkin tämän kirjan aiheiden ulkopuolella. Hieno esitys tästä tekniikasta löytyy julkaisun [KOENIG97] luvuista 6 ja 7.)

Jos vastaus kohtaan 1 on, että joillakin ZooAnimal-olioilla on tämä piirre, silloin viittauskooste on yleensä parempi (miksi uhanalaisia olioita pitäisi raahata Endangered-luokkaolion ympärillä?).

Koska Endangered-luokkaoliota ei ehkä ole, se pitää esittää osoittimella viittauksen sijasta. (Osoittimen arvoksi asetetaan 0, josta ymmärretään, että se ei viittaa mihinkään olioon. Viittauksen pitää aina kohdistua olioon. Kohdassa 3.6 selitetään tämä ero tarkemmin.)

Jos vastaus kohtaan 2 on kyllä, silloin pitää tehdä suorituksenaikaiset käsittelyfunktiot, joilla voimme lisätä ja poistaa Endangered-olion.

Esimerkissämme uhanalaisuus on joidenkin vähemmistönä olevien ZooAnimal-alityyppien ominaisuus. Lisäksi ainakin teoreettisesti se on tilanne, joka voi kääntyä päinvastaiseksi, jolloin jonain päivänä Pandaamme ei uhata enää sukupuutolla.

```
class ZooAnimal {
public:
  // ...
  const Endangered* Endangered() const;
  void addEndangered( Endangered* );
```

```
    void removeEndangered();
    // ...
protected:
    Endangered *_endangered;
    // ...
};
```

Jos sovellustamme aiotaan ajaa useilla eri alustoilla, on hyödyllistä kapseloida alustariippuvainen tieto abstraktiin luokkahierarkiaan. Sovellukseen voidaan sitten ohjelmoida alustariippumaton abstrakti rajapinta. Näyttääksemme esimerkiksi ZooAnimal-olioitamme sekä UNIX- että PC-alustoilla, voisimme määritellä DisplayManager-luokkahierarkian näin:

```
class DisplayManager{ ... };
class DisplayUnix : public DisplayManager{ ... };
class DisplayPC : public DisplayManager{ ... };
```

ZooAnimal *ei ole* eräänlainen DisplayManager, vaan *sillä on* DisplayManager-ilmentymä. ZooAnimal sisältää DisplayManager-olion koosteen eikä periytymisen kautta. Ensimmäinen kysymys on, tulisiko sen tulla arvokoosteen vai viittauskoosteen kautta?

Arvokooste ei voi esittää DisplayManager-oliota, jolla voisimme osoittaa joko todellista DisplayUnix- tai DisplayPC-oliota. Vain ZooAnimal-tietojäsenen DisplayManager-viittaus tai -osoitin mahdollistaa, että voimme käsitellä DisplayManager-alityyppiä suorituksen aikana. Tämä tarkoittaa, että vain viittauskokoonpano tukee oliokeskeistä ohjelmointia (katso julkaisusta [LIPPMAN96a] tarkempi selostus).

Toinen kysymyksemme on, kuinka voisimme päätellä, esittelemmekö ZooAnimal-jäsenen DisplayManager-viittaukseksi vai -osoittimeksi?

1. Jos todellinen DisplayManager-alityyppi on olemassa ZooAnimal-olion luontihetkellä eikä muutu ohjelman suorituksen aikana, vain silloin voidaan esitellä DisplayManager-jäsenviittaus.
2. Jos on otettu käytäntöön *laiska muistinvarausstrategia* (*lazy allocation strategy*), jossa todellista DisplayManager-alityyppiä ei varata, ennen kuin todellinen yritys näyttää olio on tehty, silloin DisplayManager-jäsen pitää esittää osoittimena ja alustaa se arvolla 0.
3. Jos haluamme vaihdella näyttötilaa suorituksen aikana, pitää myös DisplayManager-jäsen esitellä osoittimena ja alustaa se arvolla 0. *Vaihtelulla* tarkoitamme, että sallimme käyttäjän joko päättää tai vaihdella eri DisplayManager-alityyppejä ohjelman suorituksen aikana.

Tietysti käytännössä on epätodennäköistä, että sovelluksemme jokainen ZooAnimal-alioolio vaatii oman DisplayManager-alityypin, jolla näyttää itsensä. Staattinen ZooAnimal:in DisplayManager-osoitin on kaikkein todennäköisin suunnitteluvaihtoehto tässä tapauksessa.

Harjoitus 18.6

Yksilöi, mitkä seuraavista ovat tyyppiperiytymisiä ja mitkä toteutusperiytymisiä.

(a) Queue : List
(b) EncryptedString : String
(c) Gif : FileFormat
(d) Circle : Point
(e) Dqueue : Queue, List
(f) DrawableGeom : Geom, Canvas

Harjoitus 18.7

Korvaa kohdan 18.3.1 PeekbackStack-pinon Array-jäsenemme vakiokirjaston pakalla. Kirjoita pieni ohjelma, jolla voit kokeilla sitä.

Harjoitus 18.8

Etsi vastakohtia arvokoosteesta ja viittauskoosteesta. Anna esimerkki jokaisesta käyttötilanteesta, jolla kuvaat käsittelyäsi.

18.4 Luokan viittausalue periytymisessä

Jokainen luokka säilyttää oman viittausalueensa, jossa sen jäsenten ja kaikkien sisäkkäisten tyyppien nimet on määritelty (katso tarkempi käsittely kohdista 13.9 ja 13.10). Periytymisessä johdetun luokan viittausalue on sisäkkäin sen lähimmän kantaluokan viittausalueella. Jos nimeä ei ratkaista johdetun luokan viittausalueelta, etsitään nimelle määrittelyä ympäröivän kantaluokan viittausalueelta.

Tämä luokkien viittausalueiden hierarkkinen sisäkkäisyys mahdollistaa sen, että kantaluokan jäseniä voidaan käsitellä suoraan aivan kuin ne olisivat johdetun luokan jäseniä. Käykäämme läpi muutama esimerkki yksiperiytymisestä ja laajennetaan sitten käsittelyä myös moniperiytymiseen. Olkoon seuraavassa yksinkertaistettu ZooAnimal-luokkamäärittely

```
class ZooAnimal {
public:
   ostream &print( ostream& ) const;

   // julkinen vain näyttötarkoituksiin
   string is_a;
   int    ival;
private::
   double dval;
};
```

ja seuraava yksinkertaistettu johdettu Bear-luokkamäärittely

```
class Bear : public ZooAnimal {
public:
   ostream &print( ostream& ) const;
   int mumble( int );

   // julkinen vain näyttötarkoituksiin
   string name;
   int    ival;
};
```

Kun kirjoitamme

```
Bear bear;
bear.is_a;
```

on nimiresoluution todellinen prosessi seuraava:

1. bear on Bear-luokan olio. is_a:ta etsitään ensin Bear-luokan viittausalueelta. Sitä ei löydy.
2. Koska Bear on johdettu ZooAnimal:ista, etsitään ZooAnimal:in viittausalueelta seuraavaksi is_a:n esittelyä. Se löydetään ja havaitaan sen olevan ZooAnimal-kantaluokan jäsen. Viittaus ratkaistaan onnistuneesti.

Vaikka kantaluokan jäsentä voidaan käsitellä suoraan, aivan kuin se olisi johdetun luokan jäsen, käytännössä se säilyttää kantaluokan jäsenyyden. Normaalisti emme välitä, mikä luokka varsinaisesti sisältää jäsenen. Se tulee eteen silloin, kun kantaluokan ja johdetun luokan jäsenellä on sama nimi. Kun esimerkiksi kirjoitamme

```
bear.ival;
```

käsitelty ival-ilmentymä on Bear-jäsen, joka löytyi kohdan 1 etsintäprosessin kautta, kuten aikaisemmin kuvattiin.

Itse asiassa johdetun luokan jäsen, jolla on sama nimi kuin kantaluokan jäsenellä, jättää piiloon kantaluokan jäsenen suoran käsittelyn. Jotta voisimme käsitellä kantaluokan jäsentä, se pitää tarkentaa luokan viittausalueoperaattorilla:

```
bear.ZooAnimal::ival;
```

Tämä ohjaa kääntäjän etsimään ival:in esittelyä ZooAnimal-luokan viittausalueelta.

Kuvatkaamme luokan viittausalueoperaattorin käyttöä hieman naurettavalla esimerkillä (hieman naurettavalla tarkoitamme, että sinun ei tulisi koskaan tehdä tätä tuotantokoodissa!):

```
int ival;

int Bear::mumble( int ival )
{
   return ival +        // parametri-ilmentymä
      ::ival +        // globaali-ilmentymä
      ZooAnimal::ival +
      Bear::ival;
```

```
}
```

Muokkaamaton viittaus ival:iin ratkaistaan muodolliseksi parametri-ilmentymäksi. (Ellei ival:ia olisi määritelty mumble():ssa, käsiteltäisiin ival:in Bear-jäsenen ilmentymää. Ellei ival:ia olisi määritelty myös Bear-luokkaan, käsiteltäisiin ZooAnimal-jäsenen ilmentymää. Ellei ival:ia olisi määritelty myös ZooAnimal-luokkaan, käsiteltäisiin globaalia ival-ilmentymää.)

Luokan jäsenen ratkaisu suoritetaan aina ennen päättelyä, onko käsittely todella sallittua, mikä voi aluksi näyttää muulta kuin itsestään selvältä. Mietitäänpä esimerkiksi tätä uudistettua mumble()-toteutusta:

```
int dval;
int Bear::mumble( int ival )
{
   // virhe: ratkaistaan yksityiseksi ZooAnimal::dval-jäseneksi
   return ival + dval;
}
```

Voidaan väittää, että etsintäalgoritmin tulisi ratkaista ensimmäisenä löytämänsä sallitusti käsiteltävissä olevat tunnukset lähimmän tunnuksen sijaan, mutta se ei tee niin. Tässä esimerkissä etsintäalgoritmi suoritetaan seuraavasti:

1. Onko dval määritelty Bear-jäsenfunktion paikallisella viittausalueella? Ei.
2. Onko dval määritelty Bear-luokan viittausalueella? Ei.
3. Onko dval määritelty ZooAnimal-kantaluokan viittausalueella? Kyllä. Viittaus ratkaistaan tähän ilmentymään.

Nyt, kun ilmentymä on ratkaistu, kääntäjä tarkistaa, onko tuon ilmentymän käsittely sallittua. Tässä tapauksessa se ei ole: dval on yksityinen tietojäsen eikä sitä saa käsitellä suoraan mumble():ssa. Oikea (ja todennäköisesti aiottu) ratkaisu vaatii eksplisiittisen viittausalueoperaattorin:

```
return ival + ::dval; // ok
```

Pääasiallisin järkisyy, joka vaikuttaa jäsenen ratkaisuun ennen kuin mietitään sen käsittelytasoa, on estää mahdolliset hankalasti selvitettävät ohjelman käyttäytymisen muutokset, jotka perustuvat jäseneen näennäisesti liittymättömiin käsittelytasomuutoksiin. Mietipä esimerkiksi käynnistystä kuten tässä:

```
int dval;
int Bear::mumble( int ival )
{
   foo( dval );
   // ...
}
```

Jos foo() olisi ylikuormitettu funktio, ZooAnimal::dval:in siirtäminen yksityisestä jäsenestä suojatuksi jäseneksi saattaisi yhtä hyvin muuttaa koko kutsusarjaa mumble():ssa — tästä luokan suunnittelija oli täysin tietämätön, kun muutti jäsenen käsittelytasoa.

Jäsenfunktio, jolla on sama nimi ja tunniste kantaluokassa sekä johdetussa luokassa, käyttäytyy samalla tavalla kuin samanniminen tietojäsen: johdetun luokan jäsen piilottaa kantaluo-

kan jäsenen johdetun luokan viittausalueella. Jos halutaan käynnistää kantaluokan jäsen, pitää käyttää kantaluokan viittausalueoperaattoria. Esimerkiksi:

```
ostream& Bear::print( ostream &os ) const
{
   // käynnistää: ZooAnimal::print(os)
   ZooAnimal::print( os );

   os << name;
   return os;
}
```

18.4.1 Luokan viittausalue moniperiytymisessä

Kuinka moniperiytymisen esittely vaikuttaa luokan viittausalueelta etsintään? Kaikista lähikantaluokista etsitään samanaikaisesti, mikä saa aikaan moniselitteisen viittauksen mahdollisuuden, jos samanniminen jäsen on peritty kahdesta tai useammasta luokasta. Käykäämme läpi muutama esimerkki siitä, kuinka moniselitteisyys voi tulla eteen ja erilaisia strategioita sen ratkaisemiseksi. Mietipä ensiksi seuraavia luokkia:

```
class Endangered {
public:
   ostream& print( ostream& ) const;
   void highlight();
   // ...
};

class ZooAnimal {
public:
   bool onExhibit() const;
   // ...
private:
   bool highlight( int zoo_location );
   // ...
};

class Bear : public ZooAnimal {
public:
   ostream& print( ostream& ) const;
   void dance( dance_type ) const;
   // ...
};
```

Kun johdimme Panda-luokkamme käyttäen moniperiytymistä

```
class Panda : public Bear, public Endangered {
public:
   void cuddle() const;
      // ...
};
```

ja vaikka on olemassa kaksi piilevää moniselitteisyyttä sekä print()- että highlight()-funktioiden periytymisessä Bear- ja Endangered-kantaluokista, ei virheilmoitusta anneta ennen kuin moniselitteinen viittausyritys noihin funktioihin tapahtuu.

Vaikka kahden perityn print()-jäsenen moniselitteisyys on kohtuullisen selvää, kahden highlight()-jäsenen välisestä konfliktista saattaa tulla hieman yllättävä (myönnettäköön, että se sen tarkoitus on tässä). Loppujen lopuksi noilla kahdella ilmentymällä on erilaiset käsittelytasot ja funkion prototyypit. Lisäksi Endangered-ilmentymä on kahden lähimmän kantaluokan jäsen, kun taas ZooAnimal-ilmentymä on toiseksi lähimmän kantaluokan jäsen.

Se ei haittaa (no, tulemme näkemään, että itse asiassa se haittaa, mutta vain virtuaaliperiytymisessä). Bear perii ZooAnimal:in yksityisen highlight()-jäsenen; se on kirjaimellisesti näkyvä, vaikka sen käynnistystä ei sallita Bear- tai Panda-luokassa. Panda perii kaksi kirjaimellisesti näkyvää jäsentä nimeltään highlight, ja niin tarkentamaton viittaus johtaa käännösvirheeseen.

Tunnuksen etsintä alkaa lähimmältä viittausalueelta, jossa viittaus tapahtuu. Jos esimerkiksi kirjoitamme

```
int main()
{
   Panda yin_yang;
   yin_yang.dance( Bear::macarena );
}
```

lähin viittausalue, johon yin_yang kuuluu, on — Panda-luokka. Jos kirjoitamme

```
void Panda::mumble()
{
   dance( Bear::macarena );
   // ...
}
```

on lähin viittausalue paikallinen mumble()-jäsenfunktio. Jos esittely löytyy, tunnus tietysti ratkaistaan ja etsintä päättyy. Muussa tapauksessa etsitään ympäröiviltä viittausalueilta.

Moniperiytymisessä simuloidaan samanaikaista etsintää jokaisesta kantaluokan periytymispuusta — esimerkissämme sekä Endangered- että Bear/ZooAnimal-alipuista. Jos esittely löytyy jostakin näistä kantaluokan alipuista, tunnus ratkaistaan ja etsintäalgoritmi päättyy. Kun dance() käynnistetään, tapahtuu tämä:

```
// ok: Bear::dance()
yin_yang.dance( Bear::macarena );
```

Jos esittely löytyy kahdesta tai useammasta kantaluokan alipuusta, viittaus on moniselitteinen, mikä saa aikaan käännösvirheen. Kun print() käynnistetään tarkentamatta, tapahtuu tämä:

```
int main()
{
   // virhe: moniselitteinen: jompikumpi näistä
   //    Bear::print( ostream& ) const
   //    Endangered::print( ostream& ) const
   Panda yin_yang;
```

```
    yin_yang.print( cout );
}
```

Ohjelmatasoinen ratkaisu jäsenen moniselitteisyyteen on tarkentaa ilmentymä eksplisiittisesti luokan viittausalueoperaattorilla, jonka haluamme käynnistää. Esimerkiksi:

```
int main()
{
    // ok: mutta ei pidetä hyvänä ratkaisuna
    Panda yin_yang;
    yin_yang.Bear::print( cout );
}
```

Vaikka tämä ratkaisee ongelman, se ei yleensä ole riittävä ratkaisu. Syy tähän on se, että käyttäjä on asetettu asemaan, jossa hänen täytyy nyt päättää, millaista Panda-luokan oikea käyttäytyminen on. Tätä vastuutaakkaa ei tulisi koskaan sälyttää luokan käyttäjälle. Sen sijaan luokan suunnittelijan tulisi huolehtia näistä yksityiskohdista. Parempi ratkaisu on, että Panda-luokka itse ratkaisee kaikki moniselitteisyydet, jotka ovat sisäisiä sen sisäiselle hierarkialle. Yksinkertaisin tapa tehdä tämä on määritellä nimetty ilmentymä johdettuun luokkaan, joka toteuttaa halutun toiminnon. Esimerkiksi:

```
inline void Panda::highlight() {
    Endangered::highlight();
}

inline ostream&
Panda::print( ostream &os ) const
{
    Bear::print( os );
    Endangered::print( os );
    return os;
}
```

Koska moniperityn luokan esittelyn onnistunut käännös ei takaa, ettei piileviä moniselitteisyyksiä olisi, suosittelemme vahvasti, että luokan jokainen metodi tulisi tutkia moduulitestauksen aikana, vaikka se tuntuisikin merkityksettömältä.

Harjoitus 18.9

Olkoon seuraava luokkahierarkia, jossa on seuraavien tietojäsenten kokoelma

```
class Base1 {
public:
    // ...
protected:
    int    ival;
    double dval;
    char   cval;
    // ...
private:
```

```
    int    *id;
    // ...
};

class Base2 {
public:
    // ...
protected:
    float  fval;
    // ...
private:
    double dval;
    // ...
};

class Derived : public Base1 {
public:
    // ...
protected:
    string sval;
    double dval;
    // ...
};

class MI : public Derived, public Base2 {
public:
    // ...
protected:
    int          *ival;
    complex<double> cval;
    // ...
};
```

ja seuraava MI::foo()-jäsenfunktion runko

```
int ival;
double dval;

void MI::
foo( double dval )
{
    int id;
    //...
}
```

(a) Yksilöi ne jäsenet, jotka ovat näkyvissä MI:ssä. Onko yhtään näkyvissä useista kanta-luokista?

(b) Yksilöi ne jäsenet, jotka ovat näkyvissä MI::foo():sta.

Harjoitus 18.10

Käytä harjoituksessa 18.9 määriteltyä luokkahierarkiaa ja yksilöi, mitkä sijoituksista, jos yksikään, ovat virheellisiä MI::bar()-jäsenfunktiossa:

```
void MI::
bar()
{
    int sval;
  // harjoituksen kysymykset tapahtuvat täällä ...
}

(a) dval = 3.14159;(d) fval = 0;
(b) cval = 'a';    (e) sval = *ival;
(c) id = 1;
```

Harjoitus 18.11

Käytä harjoituksessa 18.9 määriteltyä luokkahierarkiaa ja seuraavaa MI::foobar()-jäsenfunktion runkoa

```
int id;

void MI::
foobar( float cval )
{
    int dval;
  // harjoituksen kysymykset tapahtuvat täällä ...
}
```

ja

(a) Sijoita paikalliseen dval-ilmentymään yhteenlasku, joka muodostuu Base1-luokan dval-jäsenestä ja Derived-luokan dval-jäsenestä.

(b) Sijoita MI-luokan cval:in reaaliosa Base2-luokan fval-jäseneen.

(c) Sijoita Base1-luokan cval-jäsen Derived-luokan sval-jäsenen ensimmäiseen merkkiin.

Harjoitus 18.12

Olkoon seuraava luokkahierarkia, jossa on seuraavat print()-nimiset jäsenfunktiot

```
class Base {
public:
    void print( string ) const;
    // ...
};

class Derived1 : public Base {
public:
```

```
    void print( int ) const;
    // ...
};

class Derived2 : public Base {
public:
    void print( double ) const;
    // ...
};

class MI : public Derived1, public Derived2 {
public:
    void print( complex<double> )const;
    // ...
};
```

(a) Miksi seuraava johtaa käännösvirheeseen?

```
MI mi;
string dancer( "Njinsky" );
mi.print( dancer );
```

(b) Kuinka voimme uudistaa MI:n määrittelyä niin, että se kääntyisi ja että se suoritettaisiin oikein?

18.5 Virtuaaliperiytyminen ◆

Oletusarvoisesti periytyminen C++:ssa on erikoismuoto arvokoosteesta. Kun kirjoitamme

```
class Bear : public ZooAnimal { ... };
```

jokainen Bear-luokkaolio sisältää kaikki sen ZooAnimal-kantaluokan aliolion ei-staattiset tietojäsenet sekä myös ei-staattiset tietojäsenet, jotka on esitelty Bear-luokassa. Samalla tavalla, kun itse johdetusta luokasta johdetaan olio, kuten

```
class PolarBear : public Bear { ... };
```

jokainen PolarBear-luokkaolio sisältää ei-staattiset tietojäsenet, jotka on esitelty PolarBear:issa sekä myös sen Bear-aliolion kaikki ei-staattiset tietojäsenet ja kaikki sen ZooAnimal-aliolion ei-staattiset tietojäsenet.

Yksiperiytymisessä arvokoosteen erikoismuoto, jota periytyminen tukee, tarjoaa mitä tehokkaimman ja tiiveimmän olion esitystavan. Siitä tulee ongelma vain moniperiytymisessä, kun kantaluokka esiintyy useita kertoja johdettujen luokkien hierarkiassa. Eräs huomattava käytännön esimerkki tästä on iostream-luokkahierarkia. Muista kuvasta 18.2, että sekä ostream- että istream-luokka on johdettu abstraktista ios-kantaluokasta. Iostream-luokka on myöhemmin johdettu sekä ostream- että istream-luokasta:

```
class iostream :
    public istream, public ostream { ... };
```

Oletusarvoisesti jokainen iostream-luokkaolio sisältää kaksi ios-alioliota: ilmentymän sen istream-aliolion sisäpuolella ja toisen ostream-aliolion sisäpuolella. Miksi tämä on huono asia? Yksinkertaisen tehokkuuden nimissä kahden ios-aliolion kopion tallentaminen tuhlaa muistia, koska iostream tarvitsee vain yhden ilmentymän. Lisäksi ios-muodostaja käynnistetään kaksi kertaa, kerran jokaiselle alioliolle. Vielä vakavampi ongelma on moniselitteisyys, jonka kaksi ilmentymää saavat aikaan. Esimerkiksi jokainen tarkentamaton ios-jäsenen käsittely johtaa käännösvirheeseen: mitä ilmentymää tarkoitetaan? Mitä, jos ostream- ja istream-luokat alustavat ios-alioliot hieman eri tavoin? Kuinka varmistumme ios-arvoparien yhtenäisyydestä iostream-luokassa? Oletusarvoisessa arvokoostemekanismissa ei ole todella hyvää tapaa varmistua siitä.

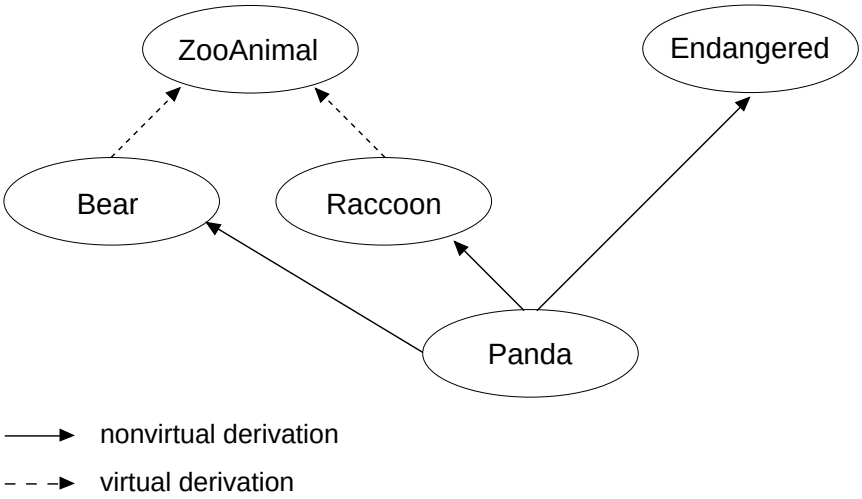
Kielen ratkaisu on tehdä vaihtoehtoinen viittauskooste periytymismekanismilla: *virtuaaliperiytyminen*. Virtuaaliperiytymisessä vain yksi, jaettu kantaluokka peritään huolimatta siitä, kuinka monta kantaluokkaa esiintyy johdettujen luokkien hierarkiassa. Jaettua kantaluokan alioliota kutsutaan *virtuaalikantaluokaksi*. Virtuaaliperiytymistä käytettäessä kantaluokan alioliioiden kopioinnin moniselitteisyys poistuu.

Koska tarkoituksenamme on käydä läpi virtuaaliperiytymisen syntaksi ja merkitys, olemme päättäneet käyttää Panda-luokkaa pedagogisena esimerkkinä. Eläintieteellisissä piireissä on yli 100 vuoden ajan silloin tällöin ollut rajuja väittelyitä siitä, kuuluuko pandakarhu (Panda) pesukarhu- (Raccoon) vai karhuheimoon (Bear). Koska ohjelmansuunnittelu on pääosin palvelualaa, on kaikkein käytännöllisin ratkaisu johtaa Panda molemmista:

```
class Panda : public Bear,
        public Raccoon, public Endangered { ... };
```

Panda-hierarkian virtuaaliperiytyminen on piirretty kuvaan 18.4, jossa kaksi pisteviivaa ilmaisevat sekä Bear:in että Raccoon:in virtuaalisen johtamisen ZooAnimal:ista, ja kolme nuolta ilmaisevat Panda:n ei-virtuaalisen johtamisen Bear:ista ja Raccoon:ista ja hyvästä vertailukoh-

dastamme, kohdan 18.2 Endangered-luokastamme.



Kuva 18.4 Panda-hierarkian virtuaaliperiytyminen

Jos tutkimme kuvaa 18.4, huomaamme yllättävän asian virtuaaliperiytymisestä: virtuaalinen johtaminen (tapauksessamme Bear ja Raccoon) pitää tehdä ennen kuin todellista tarvetta sen olemassaoloon on olemassa. Virtuaaliperiytymisestä tulee tarpeellinen, kun Panda esitellään, mutta jos Bear:ia ja Raccoon:ia ei ole vielä virtuaalisesti siinä vaiheessa johdettu, on Panda-luokan suunnittelija pulassa.

Tarkoittaako tämä, että kantaluokkamme tulisi johtaa virtuaalisesti siltä varalta, että jossain vaiheessa hierarkian elinkaaren aikana virtuaaliperiytymistä saatettaisiin tarvita? Ei. Emme missään tapauksessa suosittele sellaista. Vaikutus suorituskykyyn (ja lisäksi myöhempien johdettujen luokkien monimutkaisuuteen) voi olla merkittävä; katso suorituskyvyn mittaamisesta ja asian käsittelystä julkaisusta [LIPPMAN96a].

Tarkoitammeko, että virtuaaliperiytymistä ei tulisi koskaan käyttää? Jälleen: ei. Käytännössä kuitenkin melkein kaikissa virtuaaliperiytymisen onnistuneissa käyttötilanteissa, joissa hierarkkinen alipuu kuten iostream-kirjasto tai Panda-alipuu vaatii virtuaaliperiytymistä, kaikki on suunniteltu kerralla joko saman henkilön tai projektin suunnitteluryhmän toimesta.

Ellei virtuaaliperiytyminen tarjoa ratkaisua välittömään suunnitteluongelmaan, yleensä emme suosittele sen käyttöä. Tietysti sen lisäksi, mitä olemme sanoneet, katsotaan nyt, kuinka voisimme käyttää sitä.

18.5.1 Virtuaalisen kantaluokan esittely

Kantaluokka määritetään johdetuksi virtuaaliperiytymisen kautta muokkaamalla sen esittelyä avainsanalla virtual. Esimerkiksi seuraavat esittelyt tekevät ZooAnimal:in virtuaaliseksi kantaluokaksi sekä Bear:ille että Raccoon:ille:

```
// avainsanojen public ja virtual järjestys
// ei ole merkittävä

class Bear : public virtual ZooAnimal { ... };
class Raccoon : virtual public ZooAnimal { ... };
```

Virtuaalinen johtaminen ei ole pakonomainen asia itse kantaluokalle, vaan sen suhteelle johdettuun luokkaan. Kuten ilmoitimme aikaisemmin, virtuaaliperiytyminen toimii viittauskoosteena, eli aliolion ja sen ei-staattisten jäsenien käsittely tapahtuu epäsuorasti. Tämä tarjoaa tarvittavaa joustavuutta yhdistellä moniperiytyneitä kantaluokkien aliolioita yhteen jaettuun ilmentymään, johdettuun luokkaan. Samaan aikaan johdetun luokan oliota voidaan käsitellä osoittimella tai viittauksella kantaluokan tyyppiin, vaikka kantaluokka on virtuaalinen. Esimerkiksi kaikki seuraavan Panda-kantaluokan konversiot toimivat oikein, vaikka Panda on suunniteltu virtuaalisena periytymishierarkiana:

```
extern void dance( const Bear* );
extern void rummage( const Raccoon* );

extern ostream&
        operator<<( ostream&, const ZooAnimal& );

int main()
{
        Panda yin_yang;

        dance( &yin_yang );   // ok
        rummage( &yin_yang ); // ok
        cout << yin_yang;     // ok
        // ...
}
```

Jokainen luokka, joka voidaan määrittää kantaluokaksi, voidaan määrittää myös virtuaaliseksi kantaluokaksi, ja se voi sisältää kaikkia luokkaelementtejä, joita ei-virtuaaliset kantaluokatkin tukevat. Esimerkiksi tässä on ZooAnimal-luokkamme esittely:

```
#include <iostream>
#include <string>

class ZooAnimal;
extern ostream&
        operator<<( ostream&, const ZooAnimal& );

class ZooAnimal {
```

```
public:
        ZooAnimal( string name,
                   bool onExhibit, string fam_name )
                 : _name( name ),
                   _onExhibit( onExhibit), _fam_name( fam_name )
        {}

        virtual ~ZooAnimal();
        virtual ostream& print( ostream& ) const;
        string name() const { return _name; };
        string family_name() const { return _fam_name; }
        // ...

protected:
        bool   _onExhibit;
        string _name;
        string _fam_name;
        // ...
};
```

Lähimmän johdetun luokkailmentymän esittely ja toteutus on samanlainen kuin ei-virtuaalisesti johdetun, paitsi virtual-avainsanan käyttöä. Tässä on esimerkkinä Bear-luokkamme esittely:

```
class Bear : public virtual ZooAnimal {
public:
        enum DanceType {
            two_left_feet, macarena, fandango, waltz };

        Bear( string name, bool onExhibit=true )
           : ZooAnimal( name, onExhibit, "Bear" ),
             _dance( two_left_feet )
        {}

        virtual ostream& print( ostream& ) const;
        void dance( DanceType );
        // ...

protected:
        DanceType _dance;
        // ...
};
```

Samalla tavalla seuraavassa on Raccoon-luokkamme esittely:

```
class Raccoon : public virtual ZooAnimal {
public:
        Raccoon( string name, bool onExhibit=true )
              : ZooAnimal( name, onExhibit, "Raccoon" ),
                _pettable( false )
        {}
```

```
        virtual ostream& print( ostream& ) const;

        bool pettable() const { return _pettable;   }
        void pettable( bool petval ) { _pettable = petval; }
        // ...

protected:
        bool _pettable;
        // ...
  };
```

18.5.2 Erikoisalustuksen tulkintaa

Johdettu luokka, jossa yksi tai useampi virtuaalinen kantaluokka on epäsuorasti mukana, vaatii erikoisalustuksen tulkintaa. Käytä hetki ja vilkaise edellisen kohdan Bear- ja Raccoon-luokkien toteutuksia. Voitko yksilöidä alustusongelman, joka johtuu Panda-luokan johtamisesta?

```
class Panda : public Bear,
        public Raccoon, public Endangered {
public:
        Panda( string name, bool onExhibit=true );
        virtual ostream& print( ostream& ) const;

        bool sleeping() const { return _sleeping; }
        void sleeping( bool newval ) { _sleeping = newval; }
        // ...

protected:
        bool _sleeping;
        // ...
  };
```

Aivan oikein. Ongelma on, että sekä Bear- että Raccoon-kantaluokkien muodostajat antavat ZooAnimal-muodostajalle eksplisiittisen argumenttijoukon. Jopa pahempi asia on, että esimerkkimme argumentit eivät eroa vain heimonimeltään, vaan ne eivät kelpaa Panda-luokallemme.

Ei-virtuaalisessa johtamisessa voi johdettu luokka alustaa eksplisiittisesti vain sen lähimmät kantaluokat (katso asian käsittely kohdasta 17.4). Esimerkiksi ZooAnimal:in ei-virtuaalisessa johtamisessa Panda ei voi käynnistää ZooAnimal-muodostajaa Panda:n jäsenen alustusluettelossa. Sen sijaan virtuaalijohtamisessa vain Panda voi käynnistää suoraan sen virtuaalisen ZooAnimal-kantaluokan muodostajan.

Virtuaalisen kantaluokan alustaminen tulee *johdetuimman luokan* vastuulle. Johdetuin luokka päätellään jokaisen tietyn luokkaolion esittelystä. Kun esimerkiksi esittelemme Bear-luokkaolion, kuten tässä

```
Bear winnie( "pooh" );
```

Bear on winnie-luokkaolion johdetuin luokka ja ZooAnimal-muodostaja on se, joka suoritetaan Bear:in käynnistyksessä. Kun kirjoitamme

```
cout << winnie.family_name();
```

tulostus on

```
The family name for pooh is Bear.
```

Samalla tavalla, kun esittelemme

```
Raccoon meeko( "meeko" );
```

Raccoon on meeko-luokkaolion johdetuin luokka ja ZooAnimal-muodostaja on se, joka suoritetaan Raccoon:in käynnistyksessä. Kun kirjoitamme

```
cout << meeko.family_name();
```

tulostus on

```
The family name for meeko is Raccoon.
```

Nyt, kun esittelemme Panda-luokkaolion, kuten tässä

```
Panda yolo( "yolo" );
```

Panda on yolo-luokkaolion johdetuin luokka ja niin siitä tulee vastuullinen ZooAnimal:in alustaja.

Kun Panda-olio on alustettu, (1) ei ZooAnimal-muodostajan käynnistyksiä enää suoriteta Raccoon:issa eikä Bear:issa niiden vastaavien muodostajien suorituksen aikana ja (2) ZooAnimal-muodostaja käynnistetään argumenteilla, jotka on määritetty sille Panda-muodostajan alustusluettelossa. Tässä on toteutuksemme:

```
Panda::Panda( string name, bool onExhibit=true )
            : ZooAnimal( name, onExhibit, "Panda" ),
              Bear( name, onExhibit ),
              Raccoon( name, onExhibit ),
              Endangered( Endangered::environment,
                          Endangered::critical )
              _sleeping( false )
  {}
```

Ellei Panda-muodostajassa ole määritetty argumentteja eksplisiittisesti ZooAnimal-muodostajalle, tapahtuu jompikumpi toimenpide: kutsutaan joko ZooAnimal-oletusmuodostajaa tai, ellei oletusmuodostajaa ole, kääntäjä antaa virheilmoituksen, kun Panda-muodostajan määrittelyä käännetään.

Kun kirjoitamme

```
cout << yolo.family_name();
```

tulostus on

```
The family name for yolo is Panda.
```

Panda-luokassa sekä Bear- että Raccoon-luokat toimivat pikemminkin väliluokkina kuin johdetuimpina luokkina. Johdetulle väliluokalle supistetaan kaikkien virtuaalisten kantaluok-

kien muodostajien suoria käynnistyksiä automaattisesti. Jos Panda johdetaan myöhemmin, tulee siitä itsestä johdetun luokkaolion väliluokka ja sen ZooAnimal-muodostajan käynnistystä supistetaan automaattisesti.

Olet varmaan huomannut, että ne kaksi argumenttia, jotka välitetään sekä Bear- että Raccoon-muodostajille, ovat tarpeettomia, kun luokat toimivat johdettuina väliluokkina. Suunnitteluratkaisu, jolla voidaan välttää tarpeeton argumenttivälitys, on tehdä eksplisiittinen muodostaja, joka käynnistetään, kun luokka toimii johdettuna väliluokkana. Esimerkiksi Bear-väliluokkamme muodostajaa voitaisiin muokata seuraavasti:

```
class Bear : public virtual ZooAnimal {
public:
        // kun on johdetuin luokka
        Bear( string name, bool onExhibit=true )
           : ZooAnimal( name, onExhibit, "Bear" ),
             _dance( two_left_feet )
        {}

        // ... loput samalla tavalla

protected:
        // kun on johdettu väliluokka
        Bear() : _dance( two_left_feet ) {}

        // ... loput samalla tavalla
  };
```

Suunnittelemme tämän muodostajailmentymän suojatuksi, koska tarkoitus on, että se käynnistetään vain myöhemmin johdettavista luokista. Olettaen, että teemme samanlaisen oletusmuodostajan Raccoon:ille, voimme nyt uudistaa Panda-muodostajamme seuraavasti:

```
Panda::Panda( string name, bool onExhibit = true )
            : ZooAnimal( name, onExhibit, "Panda" ),
              Endangered( Endangered::environment,
                          Endangered::critical )
              _sleeping( false )
  {}
```

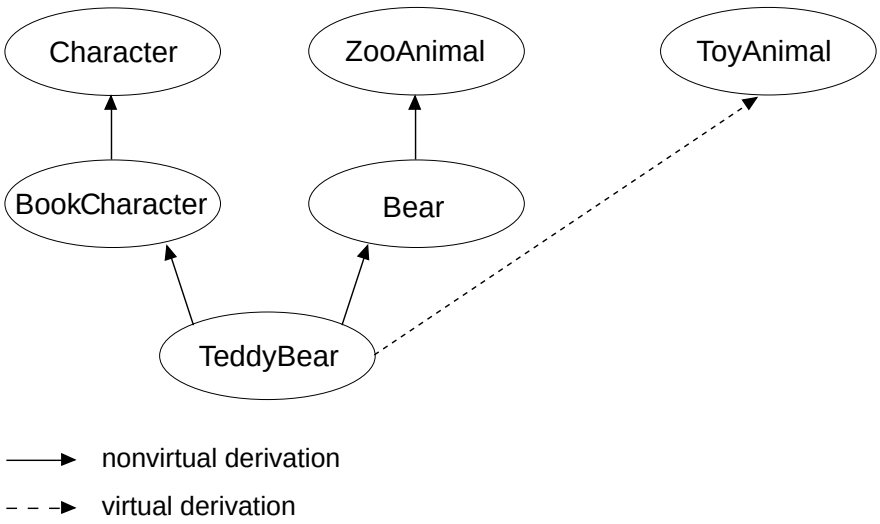
18.5.3 Muodostaja- ja tuhoajajärjestys

Virtuaaliset kantaluokat muodostetaan aina ennen ei-virtuaalisia kantaluokkia huolimatta siitä, missä ne esiintyvät periytymishierarkiassa. Esimerkiksi seuraavassa — myönnettäköön, eriskummallisessa — johdetussa TeddyBear-luokassa on kaksi virtuaalista kantaluokkaa: lähin on ToyAnimal-ilmentymä ja ZooAnimal-ilmentymä, josta Bear on johdettu:

```
class Character { ... };
class BookCharacter : public Character { ... };
class ToyAnimal { ... };

class TeddyBear : public BookCharacter,
        public Bear, public virtual ToyAnimal
        { ... };
```

Hierarkia on piirretty kuvaan 18.5, jossa virtuaalijohtamiset on ilmaistu pisteviivoilla ja ei-virtuaaliset johtamiset yhtenäisillä viivoilla.



Kuva 18.5 TeddyBear-hierarkian virtuaaliperiytyminen

Lähimmät kantaluokat tutkitaan siinä järjestyksessä, jossa ne on esitelty virtuaalisissa kantaluokissa. Esimerkissämme tutkitaan ensin BookCharacter:in periytymisen alipuu, sitten Bear:in ja lopuksi ToyAnimal:in. Jokainen alipuu tutkitaan syvyydeltään ensin; tarkoittaa, että etsintä alkaa juuriluokasta ja siirtyy alas. BookCharacter-alipuussa tutkitaan ensin Character, sitten BookCharacter. Bear-alipuussa tutkitaan ensin ZooAnimal, sitten Bear.

TeddyBear:in virtuaalisten kantaluokkien muodostajien käynnistysjärjestys tällä etsintäjärjestyksellä on ZooAnimal ensin, sitten ToyAnimal.

Kun virtuaalisten kantaluokkien muodostajat on käynnistetty, ei-virtuaalisten kantaluokkien

muodostajat käynnistetään niiden esittelyjärjestyksessä: BookCharacter, sitten Bear. Ennen BookCharacter-muodostajan suoritusta käynnistetään sen kantaluokan Character-muodostaja.

Esittelyssä

```
TeddyBear Paddington;
```

on kantaluokkien muodostajien järjestys seuraava:

```
ZooAnimal();     // Bear:in virtuaalinen kantaluokka
ToyAnimal();     // lähin virtuaalinen kantaluokka
Character();     // BookCharacter:in ei-virtuaalinen kantaluokka
BookCharacter();// lähin ei-virtuaalinen kantaluokka
Bear();          // lähin ei-virtuaalinen kantaluokka
TeddyBear();     // johdetuin luokka
```

jossa ZooAnimal:in ja ToyAnimal:in alustukset ovat TeddyBear:in vastuulla, joka on Paddington-luokkaolion johdetuin luokka.

Kopiointimuodostajien käynnistysjärjestys jäsenittäisessä alustuksessa (ja kopioinnin sijoitusoperaattoreilla jäsenittäisessä sijoituksessa) on sama. On taattu, että kantaluokkien tuhoajien kutsujärjestys on päinvastainen kuin muodostajien käynnistysjärjestys.

18.5.4 Virtuaalisen kantaluokan jäsenien näkyvyys

Määritelkäämme uudelleen Bear-luokkamme ja tehkäämme sille oma ilmentymä ZooAnimal:in onExhibit()-jäsenfunktiosta:

```
bool Bear::onExhibit() { ... }
```

Kun onExhibit()-jäsenfunktioon viitataan Bear-luokkaolion kautta, se ratkaistaan Bear:in ilmentymäksi:

```
Bear winnie( "a lover of honey" );
winnie.onExhibit(); // Bear::onExhibit()
```

Kun onExhibit()-jäsenfunktioon viitataan Raccoon-luokkaolion kautta, se ratkaistaan silti perityksi ZooAnimal-jäseneksi:

```
Raccoon meeko( "a lover of all foods" );
meeko.onExhibit(); // ZooAnimal::onExhibit()
```

Johdettu Panda-luokka perii kahden kantaluokkansa jäsenet. Nämä jakautuvat kolmeen kategoriaan:

1. Virtuaalisen ZooAnimal-kantaluokan ilmentymät kuten name() ja family_name(), joita eivät kumoa johdetut luokat Bear tai Raccoon.
2. Virtuaalisen ZooAnimal-kantaluokan onExhibit()-ilmentymä, joka on peritty Raccoon:in kautta ja joka kumoaa Bear:in määrittelemän ilmentymän.
3. ZooAnimal:in print()-funktion erikoistetut Bear- ja Raccoon-ilmentymät.

Mitä perityistä jäsenistä voidaan käsitellä ilman moniselitteisyyttä ja suoraan Panda-luokan viittausalueelta? Ei-virtuaalisessa johtamisessa vastaus on, että ei mitään. Kaikki tarkentamattomat viittaukset ei-virtuaalisessa johtamisessa ovat moniselitteisiä. Virtuaalijohtamisessa kaikkia kohtien 1 ja 2 jäseniä voidaan käsitellä ilman moniselitteisyyttä ja suoraan. Esimerkkinä seuraava Panda-luokkaolio

```
Panda spot( "Spottie" );
```

Kutsu

```
spot.name();
```

käynnistää jaetun virtuaalisen ZooAnimal-kantaluokan `name()`-jäsenfunktion. Kutsu

```
spot.onExhibit();
```

käynnistää johdetun Bear:in `onExhibit()`-jäsenfunktion.

Kun kaksi tai useampi jäsenen ilmentymä on peritty eri johtamispolkujen kautta (tämä ei päde ainoastaan jäsenfunktioihin, vaan myös tietojäseniin ja sisäkkäisiin tyyppeihin) ja jos molemmat edustavat saman virtuaalisen kantaluokan jäsentä, ei moniselitteisyyttä ole, koska jäsenen yksi ilmentymä on jaettu (kohta 1); jos yksi edustaa virtuaalisen kantaluokan jäsentä ja toinen myöhemmin johdettua ilmentymää, moniselitteisyyttä ei ole (johdetun luokan erikoistetulle ilmentymälle annetaan parempi sidontajärjestys kuin virtuaalisen kantaluokan jaetulle ilmentymälle [kohta 2]); mutta, jos molemmat edustavat myöhemmin johdetun luokan ilmentymiä, on jäsenen suora käsittely moniselitteinen. Tämä ratkaistaan parhaiten tekemällä kumoava ilmentymä johdettuun luokkaan (kohta 3).

Esimerkiksi ei-virtuaalisessa johtamisessa tarkentamaton viittaus `onExhibit()`:iin Panda-luokkaolion kautta on moniselitteinen.

```
// virhe: moniselitteinen ei-virtuaalisessa johdannaisuudessa
Panda yolo( "a lover of bamboo" );
yolo.onExhibit();
```

Ei-virtuaalisessa johtamisessa jokaiselle peritylle ilmentymälle annetaan sama painoarvo viittausta ratkaistaessa, jolloin tarkentamaton viittaus johtaa käännöksenaikaiseen moniselitteisyysvirheeseen (katso aiheen käsittely kohdasta 18.4.1).

Virtuaalijohtamisessa virtuaalisen kantaluokan jäsenen periytymiselle annetaan vähemmän painoarvoa kuin myöhemmin tuon jäsenen uudelleenmääritellylle ilmentymälle. Peritylle `onExhibit()`:in Bear-ilmentymälle annetaan korkeampi sidontajärjestys kuin ZooAnimal-ilmentymälle, joka on peritty Raccoon:in kautta:

```
// ok: ei moniselitteinen virtuaaliperiytymisessä
// Bear::onExhibit() käynnistetään
yolo.onExhibit();
```

Jos kaksi tai useampi kantaluokka samalla johdetulla tasolla määrittelee uudelleen virtuaalisen kantaluokan jäsenen, niille myönnetään yhtäläinen sidontajärjestys johdetussa luokassa. Jos esimerkiksi Raccoon myös määrittelisi `onExhibit()`-jäsenen, pitäisi Panda:n tarkentaa jokai-

nen käsittelytilanne sopivalla luokan viittausalueoperaattorilla.

```
bool Panda::onExhibit()
{
    return Bear::onExhibit() &&
                Raccoon::onExhibit() &&
                ! _sleeping;
}
```

Harjoitus 18.13

Olkoon seuraava luokkahierarkia:

```
class Class { ... };
class Base : public Class { ... };
class Derived1 : virtual public Base { ... };
class Derived2 : virtual public Base { ... };
class MI : public Derived1,
        public Derived2 { ... };
class Final : public MI, public Class { ... };
```

(a) Mikä on Final-luokkaolion määrittelyn muodostajan ja tuhoajan järjestys?

(b) Kuinka monta Base-kantaoliota Final-luokkaolio sisältää? Kuinka monta Class-alioliota?

(c) Mitkä seuraavista sijoituksista johtavat käännösvirheeseen?

```
Base    *pb;
MI      *pmi;
Class   *pc;
Derived2 *pd2;

(i)  pb = new Class;  (iii) pmi = pb;
(ii) pc = new Final;  (iv)  pd2 = pmi;
```

Harjoitus 18.14

Olkoon seuraava luokkahierarkia ja sen jäsenet

```
class Base {
public:
   bar( int );
   // ...
protected:
   int ival;
   // ...
};

class Derived1 : virtual public Base {
public:
   bar( char );
   foo( char );
```

```
    // ...
protected:
    char cval;
    // ...
};

class Derived2 : virtual public Base {
public:
    foo( int );
    // ...
protected:
    int  ival;
    char cval;
    // ...
};

class VMI : public Derived1, public Derived2 {};
```

Mitä perittyjä jäseniä voidaan käsitellä VMI-luokasta ilman tarkennusta? Mitkä vaativat tarkennuksen?

Harjoitus 18.15

Olkoon seuraava Base-luokka ja kolme muodostajaa

```
class Base {
public:
    Base();
    Base( string );
    Base( const Base& );
    // ...
protected:
    string _name;
};
```

Määrittele vastaavat kolme muodostajaa seuraaville:

(a) Jompikumpi:
```
    class Derived1 : virtual public Base{ ... };
    class Derived2 : virtual public Base{ ... };
```
(b) `class VMI : public Derived1, public Derived2{ ... };`

(c) `class Final : public VMI{ ... };`

18.6 Esimerkki moni- ja virtuaaliperiytymisestä ◆A

Tässä kohdassa kuvaamme monivirtuaalihierarkian määrittelyä ja käyttöä toteuttamalla kohdassa 2.4 esitellyn Array-luokkamallin hierarkian. Toteutuksemme perustuu luvussa 16 esitettyyn Array-luokkamalliin, mutta olemme muokanneet sitä toimimaan konkreettisena kantaluokkana. Toteutusta ennen kerromme lyhyesti luokkamallien käytöstä periytymisessä.

Luokkamallin ilmentymä voi toimia kuin eksplisiittinen kantaluokka, kuten seuraavassa:

```
class IntStack : private Array<int> {};
```

Vaihtoehtoisesti luokkamalli voidaan johtaa mallittomasta kantaluokasta, kuten seuraavassa:

```
class Base {};
template < class Type >
   class Derived : public Base {};
```

tai toimia johtamisessa sekä johdettuna että kantaluokkana:

```
template < class Type >
   class Array_RC : public virtual Array<Type> {};
```

Ensimmäisessä esimerkissä Array-luokkamallin kokonaislukuinstantiointi toimii mallittoman IntStack-luokan yksityisenä kantaluokkana. Toisessa esimerkissä malliton Base-luokka toimii jokaisen Derived-luokkamallin instantioinnin kantaluokkana. Kolmannessa esimerkissä jokaisella Array_RC-luokkamallin instantioinnilla on vastaava Array-luokkamallin instantiointi kantaluokkanaan. Esimerkiksi

```
Array_RC<int> ia;
```

generoi kokonaislukuilmentymän sekä Array- että Array_RC-luokkamalleista.

Lisäksi itse malliparametri voi toimia kantaluokkana. Esimerkiksi [MURRAY93] kuvaa tätä tapausta

```
template < typename Type >
   class Persistent : public Type{ ... };
```

joka määrittelee johdetun, pysyvän alityypin jokaiselle instantioidulle tyypille. Kuten Murray huomauttaa, `Type`:n implisiittinen rajoitus on, että sen pitää olla luokkatyyppi. Esimerkiksi:

```
Persistent< int > pi; // hups, virhe
```

johtaa käännösvirheeseen, koska sisäiset tyypit eivät voi olla johtamisen olioina.

Kun luokkamalli toimii kantaluokkana, se pitää tarkentaa sen täydellisellä parametriluettelolla. Jos on esimerkiksi seuraava luokkamallin määrittely

```
template < class T > class Base {};
```

niin voidaan kirjoittaa

```
template < class Type >
    class Derived : public Base<Type> {};
```

eikä

```
// virhe: Base on malli
// eikä malliargumentteja saa määrittää
template < class Type >
    class Derived : public Base{};
```

Seuraavassa kohdassa luvussa 16 määritelty Array-luokkamalli toimii virtuaalisena kantaluokkana (1) raja-arvoja tarkistavalle Array-alityypille, (2) lajitellulle Array-alityypille ja (3) Array-alityypille, joka sekä on lajiteltu että tukee raja-arvotarkistusta. Alkuperäinen Array-luokkamallin toteutus on kuitenkin sopimaton tähän johtamiseen:

- Kaikki sen tietojäsenet ja apufunktiot ovat yksityisiä, eivät suojattuja.
- Yhtäkään tyyppiriippuvaisista funktioista, kuten indeksioperaattoria, ei ole määritetty virtuaaliseksi.

Tarkoittaako tämä, että alkuperäinen toteutuksemme oli väärin? Ei. Se oli oikein niin kauan kuin ymmärsimme sitä. Alkuperäisen Array-luokkamallin toteutuksen aikaan emme osanneet tarvita erikoistettuja Array-alityyppejä. Nyt, kun osaamme, pitää Array-luokkamallin määrittelyä uudistaa (jäsenfunktion toteutus säilyy ennallaan). Tässä on uusi Array-luokkamallimme määrittely:

```
#ifndef ARRAY_H
#define ARRAY_H

#include <iostream>

// tarvitaan operator<<-operaattorin edelleen esittelyyn
template <class Type> class Array;

template <class Type> ostream&
   operator<<( ostream&, const Array<Type>& );

template <class Type>
class Array {
   static const int ArraySize = 12;
public:
   explicit Array( int sz = ArraySize ) { init( 0, sz ); }
   Array( const Type *ar, int sz )      { init( ar, sz); }
   Array( const Array &iA )     { init( iA.ia, iA.size()); }
   virtual ~Array()             { delete [] ia; }

   Array& operator=( const Array& );
   int size() { return _size; }
   virtual void grow();
```

```
    virtual void print( ostream& = cout );

    Type at( int ix ) const { return ia[ ix ]; }
    virtual Type& operator[]( int ix ) { return ia[ix]; }

    virtual void sort( int,int );
    virtual int find( Type );
    virtual Type min();
    virtual Type max();

protected:
    void swap( int,int );
    void init( const Type*, int );

    int _size;
    Type *ia;
};

#endif
```

Eräs ongelma tässä siirry eteenpäin -monimuotoisuudessa on, että yleinen indeksioperaattorin käyttö on siirtynyt välittömästä muistinkäsittelystä huomattavan raskaaseen virtuaalifunktiokutsuun. Esimerkiksi seuraavassa funktiossa, mihin tahansa tyyppiin `ia` viittaakin, vain yksinkertainen välitön elementin luku on tarpeen:

```
int find( const Array< int > &ia, int value )
{
    for ( int ix = 0; ix < ia.size(); ++ix )
        // nyt virtuaalisen funktion kutsu
        if ( ia[ ix ] == value )
            return ix;
    return -1;
}
```

Mukautuaksemme suorituskykyhuoliin, teemme välittömän `at()`-jäsenfunktion, jolla päästää lukemaan elementti suoraan.

18.6.1 Raja-arvot tarkistavan Array-luokan johtaminen

Kohdan 16.13 `try_array()`-funktiossa, jota käytettiin aikaisemman Array-luokkamallin toteutuksen kokeiluun, oli kaksi lausetta:

```
int index = iA.find( find_val );
Type value = iA[ index ];
```

find() palauttaa ensimmäisen find_val-ilmentymän indeksin tai -1, ellei arvoa löydy taulukosta. Tämä koodi on virheellinen, koska se ei testaa mahdollista paluuarvoa -1. Koska -1 menee taulukon alueen ulkopuolelle, on jokainen alustusarvo mahdollisesti kelpaamaton ja ohjelmamme jokainen suoritus on mahdollisesti virheellinen. Määritelkäämme raja-arvot tarkistava Array-alityyppi. Kutsumme sitä nimellä Array_RC ja määrittelemme sen otsikkotiedostoon nimeltään Array_RC.h:

```
#ifndef ARRAY_RC_H
#define ARRAY_RC_H

#include "Array.h"

template <class Type>
class Array_RC : public virtual Array<Type> {
public:
   Array_RC( int sz = ArraySize )
            : Array<Type>( sz ) {}

   Array_RC( const Array_RC& r );
   Array_RC( const Type *ar, int sz );
   Type& operator[]( int ix );
};

#endif
```

Johdetun luokan määrittelyssä jokainen viittaus kantaluokkamallin tyyppimääreeseen pitää tarkentaa sen muodollisella parametriluettelolla. Pitää kirjoittaa

```
Array_RC( int sz = ArraySize )
     : Array<Type>( sz ) {}
```

eikä

```
// virhe: Array ei ole tyyppimääre
Array_RC( int sz = ArraySize ) : Array( sz ) {}
```

Array_RC-luokan ainoa erikoistunut piirre on raja-arvojen tarkistus, jonka tekee sen indeksioperaattori. Muussa tapauksessa Array-luokkamallin toteutusta voidaan käyttää uudelleen sellaisenaan. Koska muodostajia *ei* kuitenkaan peritä, muista, että Array_RC-luokka määrittelee kolme muodostajaa. Array_RC:n virtuaalinen johtaminen Array:sta ennakoi myöhempää monijohtamista, jota katsomme myöhemmin.

Tässä on koko Array_RC-jäsenfunktion toteutus sijoitettuna tiedostoon nimeltään Array_RC.C (Array-funktion määrittelyt on sijoitettu Array.C-otsikkotiedostoon, koska käytämme malli-instantioinnissa mukaan ottavaa mallia [katso asian käsittely kohdasta 16.18.1]):

```
#include "Array_RC.h"
#include "Array.C"
#include <assert.h>

template <class Type>
Array_RC<Type>::Array_RC( const Array_RC<Type> &r )
    : Array<Type>( r ) {}

template <class Type>
Array_RC<Type>::Array_RC( const Type *ar, int sz )
    : Array<Type>( ar, sz ) {}

template <class Type>
Type &Array_RC<Type>::operator[]( int ix ) {
    assert( ix >= 0 && ix < Array<Type>::_size );
    return ia[ ix ];
}
```

Miksi tarkennamme viitatut Array-kantaluokan jäsenet, kuten seuraavassa _size:n kohdalla?

```
Array<Type>::_size;
```

Niin pitää tehdä, jotta voimme estää Array-kantaluokan tutkimisen siihen saakka, kunnes malli on instantioitu. Teemme sen asettamalla viittauksen riippuvaiseksi malliparametrista. Tämä tarkoittaa, että nimet Array_RC-määrittelyssä (paitsi ne, jotka riippuvat eksplisiittisesti malliparametrista) ratkaistaan, kun malli on määritelty. Kun tarkentamatonta _size-nimeä käytetään, kääntäjän pitää löytää määrittely _size:lle, ellei nimi riipu eksplisiittisesti malliparametrista. _size-nimi on tehty riippuvaiseksi malliparametrista laittamalla sen eteen kantaluokan nimi, Array<Type>. Kääntäjä ei siten yritä ratkaista _size-nimeä ennen kuin malli on instantioitu. (Näemme lisää esimerkkejä tästä Array_Sort-luokan määrittelyssä.)

Jokainen Array_RC:n instantiointi generoi vastaavan Array-luokan ilmentymän. Esimerkiksi

```
Array_RC<string> sa;
```

generoi sekä string Array_RC- että String Array -ilmentymän. Seuraava ohjelma ajaa uudelleen try_array():n (katso toteutus kohdasta 16.13) ja välittää sille Array_RC-alityypin oliot. Jos toteutuksemme on oikein, saadaan raja-arvojen ylitys siepattua.

```
#include "Array_RC.C"
#include "try_array.C"

main()
{
   static int ia[10] = { 12,7,14,9,128,17,6,3,27,5 };
   Array_RC<int> iA( ia,10 );
```

```
    cout << "class template instantiation Array_RC<int>\n";
    try_array( iA );

    return 0;
}
```

Kun ohjelma käännetään ja suoritetaan, se generoi seuraavan tulostuksen:

```
class template instantiation Array_RC<int>

try_array: initial array values:
( 10 )< 12, 7, 14, 9, 128, 17
      6, 3, 27, 5 >

try_array: after assignments:
( 10 )< 128, 7, 14, 9, 128, 128
      6, 3, 27, 3 >

try_array: memberwise initialization
( 10 )< 128, 7, 14, 9, 128, 128
      6, 3, 27, 3 >

try_array: after memberwise copy
( 10 )< 128, 7, 128, 9, 128, 128
      6, 3, 27, 3 >

try_array: after grow
( 16 )< 128, 7, 128, 9, 128, 128
      6, 3, 27, 3, 0, 0
      0, 0, 0, 0 >

value to find: 5index returned: -1
Assertion failed: ix >= 0 && ix < _size
```

18.6.2 Lajitellun Array-luokan johtaminen

Tässä on toinen Array-luokkamme erikoistaminen — lajiteltu Array-alityyppi. Kutsumme sitä nimellä Array_Sort ja määrittelemme sen otsikkotiedostoon nimeltään Array_S.h:

```
#ifndef ARRAY_S_H
#define ARRAY_S_H

#include "Array.h"

template <class Type>
class Array_Sort : public virtual Array<Type> {
protected:
    void set_bit()   { dirty_bit = true; }
    void clear_bit() { dirty_bit = false; }
```

```
    void check_bit() {
     if ( dirty_bit ) {
          sort( 0, Array<Type>::_size-1 );
          clear_bit();
     }
    }

public:
    Array_Sort( const Array_Sort& );
    Array_Sort( int sz = Array<Type>::ArraySize )
       : Array<Type>( sz )
         { clear_bit();  }

    Array_Sort( const Type* arr, int sz )
        : Array<Type>( arr, sz )
        { sort( 0,Array<Type>::_size-1 ); clear_bit(); }

    Type& operator[]( int ix )
       { set_bit(); return ia[ ix ]; }

    void print( ostream& os = cout )
      { check_bit(); Array<Type>::print( os ); }

    Type min() { check_bit(); return ia[ 0 ]; }
    Type max() { check_bit(); return ia[ Array<Type>::_size-1 ]; }

    bool is_dirty() const { return dirty_bit; }
    int find( Type );
    void grow();

protected:
    bool dirty_bit;
};

#endif
```

Array_Sort esittelee lisätietojäsenen dirty_bit. Jos dirty_bit on asetettu, ei enää taata, että se on lajitellussa järjestyksessä. Mukana on useita tukea antavia käsittelyfunktioita: is_dirty() palauttaa dirty_bit-olion arvon, set_bit() sijoittaa dirty_bit-olioon arvon tosi, clear_bit() sijoittaa dirty_bit-olioon arvon epätosi ja check_bit() lajittelee taulukon uudelleen, jos dirty_bit on asetettu arvoon tosi, ja tyhjentää sitten dirty_bit-olion. Jokainen operaatio, joka mahdollisesti laittaa taulukon epäjärjestykseen, käynnistää set_bit():in.

Jokaisen viittauksen Array-kantaluokkamalliin pitää määrittää luokan koko parametriluettelo.

```
Array<Type>::print( os );
```

käynnistää kantaluokan print()-funktion jokaiselle Array_Sort-instantioinnin vastaavalle Array-luokan ilmentymälle. Esimerkiksi

```
Array_Sort<string> sas;
```

instantioi sekä string Array_Sort - että string Array -luokkailmentymän.

```
cout << sas;
```

instantioi tulostusoperaattorin string Array -ilmentymän, jolle sas välitetään. Kutsu

```
ar.print( os );
```

käynnistää operaattorissa string Array_Sort -ilmentymän virtuaalisen print()-ilmentymän. Aluksi käynnistetään check_bit(). Kun se on tehty, käynnistetään print()-funktion string Array -ilmentymä staattisesti. (Muistathan, että staattisesti käynnistäminen tarkoittaa, että funktio ratkaistaan käännöksen aikana ja laajennetaan välittömästi, jos se on tarkoituksenmukaista.) Virtuaalifunktio, joka perustuu ar:n osoittamaan todelliseen olioon, käynnistetään tavallisesti dynaamisesti suorituksen aikana. Virtuaalimekanismi kumoutuu, kun virtuaalifunktio käynnistetään eksplisiittisesti käyttäen luokan viittausalueoperaattoria kuten tässä: Array::print(). Tämä on tärkeä tehokkuuden apu, kun käynnistämme kantaluokan eksplisiittisen virtuaalifunktion ilmentymän tuon funktion johdetussa luokkailmentymässä, kuten teemme print():in Array_Sort-ilmentymässä. Katso aiheen käsittely kohdasta 17.5.)

Jäsenfunktiot, jotka on määritelty luokkamäärittelyn ulkopuolelle, on sijoitettu tiedostoon nimeltään Array_S.C. Esittely voi näyttää hälyttävän monimutkaiselta, mikä johtuu mallisyntaksista. Paitsi parametriluetteloa, esittely on kuitenkin samanlainen mallittomalle luokalle:

```
template <class Type>
Array_Sort<Type>::
Array_Sort( const Array_Sort<Type> &as )
   : Array<Type>( as )
{
   // huomaa: as.check_bit() ei toimi!
   // -- katso selitystä tuonnempana ...
   if ( as.is_dirty() )
       sort( 0, Array<Type>::_size-1 );
   clear_bit();
}
```

Jokainen mallin nimen käyttötilanne tyyppimääreenä pitää tarkentaa sen koko parametriluettelolla. Kirjoitamme

```
template <class Type>
Array_Sort<Type>::
Array_Sort( const Array_Sort<Type> &as )
```

emmekä

```
template <class Type>
Array_Sort<Type>::
Array_Sort<Type>( // virhe: ei ole tyyppimääre
```

koska toinen Array_Sort-esiintymä toimii funktion nimenä *eikä* tyyppimääreenä.

Syy siihen, että kirjoitamme

```
if ( as.is_dirty() )
    sort( 0, _size );
```

emmekä

```
as.check_bit();
```

on kaksinainen. Ensimmäinen syy on tyyppiturvallisuus: check_bit() ei ole const-jäsenfunktio — se muokkaa siihen liittyvää luokkaoliota. Argumentti as välitetään viittauksena vakio-oliolle. Kun kutsutaan check_bit():iä, se rikkoo sen const-tyyppisyyttä ja saa aikaan käännösvirheen.

Toinen syy on se, että kopiointimuodostaja ei huolestu as-argumenttiin liittyvästä taulukosta muuten, kuin että se päättelee, tarvitseeko juuri luotu Array_Sort-olio lajitella. Muista, että uuden Array_Sort-olion uusi dirty_bit-tietojäsen on vielä alustamatta. Kun Array_Sort-muodostajan runko aloittaa suorituksen, vain Array-luokasta perityt ia- ja _size-jäsenet on vasta alustettu. Array_Sort-muodostajan pitää sekä alustaa sen lisätietojäsenet (kutsumalla clear_bit()-funktiota) että pakottaa sen alityyppi tekemään kaikki erikoistetut piirteet (kutsumalla sort()-funktiota). Vaihtoehtoinen Array_Sort-muodostajan toteutus voisi olla seuraava:

```
// vaihtoehtoinen toteutus
template <class Type>
Array_Sort<Type>::
Array_Sort( const Array_Sort<Type> &as )
   : Array<Type>( as )
{
   dirty_bit = as.dirty_bit;
   check_bit();
}
```

Tässä on grow()-jäsenfunktion toteutus[1]. Strategia on käyttää perittyä Array-luokan ilmentymää uudelleen lisämuistin varaamiseen, sitten lajitella taulukon elementin uudelleen ja tyhjentää dirty_bit:

```
template <class Type>
void Array_Sort<Type>::grow()
{
   Array<Type>::grow();
   sort( 0, Array<Type>::_size-1 );
   clear_bit();
}
```

Tässä on binäärietsintätoteutus Array_Sort:in find()-ilmentymästä:

```
template <class Type>
```

1. Tähän liittyy potentiaalinen vaara “roikkuvasta” viittauksesta, jos asiakas on tallentanut elementin osoitteen, joka on palautettu alkuperäisestä taulukosta viittauksena, ennen kuin grow() kopioi elementin uuteen paikkaan. Nähdäksesi aiheen koko käsittelyn, katso Tom Cargillin artikkeli julkaisusta [LIPPMAN96b].

```
int Array_Sort<Type>::find( Type val )
{
    int low = 0;
    int high = Array<Type>::_size-1;
    check_bit();

    while ( low <= high ) {
        int mid = ( low + high )/2;
        if ( val == ia[ mid ])
            return mid;
        if ( val < ia[ mid ] )
            high = mid-1;
        else low = mid+1;
    }
    return -1;
}
```

Yritetäänpä Array_Sort-luokan toteutusta try_array()-funktiota käyttämällä. Seuraava ohjelma testaa Array_Sort-luokan sekä kokonaisluku- että string-instantiointia.

```
#include "Array_S.C"
#include "try_array.C"
#include <string>

main()
{
   static int ia[ 10 ] = { 12,7,14,9,128,17,6,3,27,5 };
   static string sa[ 7 ] = {
              "Eeyore", "Pooh", "Tigger",
              "Piglet", "Owl", "Gopher", "Heffalump"
   };

   Array_Sort<int> iA( ia,10 );
   Array_Sort<string> SA( sa,7 );

   cout << "class template instantiation Array_Sort<int>"
       << endl;
   try_array( iA );

   cout << "class template instantiation Array_Sort<string>"
       << endl;
   try_array( SA );

   return 0;
}
```

Kun ohjelma käännetään ja suoritetaan, string-ilmentymän tulostus näyttää seuraavalta — huomaa, että tämä epäonnistuu suorituksessa, jos yritetään näyttää elementti indeksin arvolla, joka on raja-arvojen ulkopuolella eli arvolla -1.

```
class template instantiation Array_Sort<string>

try_array: initial array values:
( 7 )< Eeyore, Gopher, Heffalump, Owl, Piglet, Pooh
     Tigger >

try_array: after assignments:
( 7 )< Eeyore, Gopher, Owl, Piglet, Pooh, Pooh
     Pooh >

try_array: memberwise initialization
( 7 )< Eeyore, Gopher, Owl, Piglet, Pooh, Pooh
     Pooh >

try_array: after memberwise copy
( 7 )< Eeyore, Piglet, Owl, Piglet, Pooh, Pooh
     Pooh >

try_array: after grow
( 11 )< <empty>, <empty>, <empty>, <empty>, Eeyore, Owl
     Piglet, Piglet, Pooh, Pooh, Pooh >

value to find: Tiggerindex returned: -1
Memory fault(coredump)
```

Huomaa, että jäsenittäin kopioidun Array-luokan string-ilmentymän tulostusta *ei ole* lajiteltu. Miksi? Koska virtuaalifunktio on käynnistetty luokkaolion kautta eikä osoittimen tai viittauksen kautta. Kuten kerroimme kohdassa 17.5, kun ilmentymä käynnistetään luokkaolion kautta, se vaikuttaa tuon olion luokkatyypin aktiiviseen virtuaalifunktioon, ei olion luokkatyyppiin, joka on voitu sijoittaa siihen. Täten Sort-ilmentymää ei koskaan käynnistetä Array-luokkaolion kautta. (Otamme tähän mukaan yksinkertaisen, havainnollisen esimerkin. Emme tekisi niin todellisessa tuotantokoodissa.)

18.6.3 Monijohdettu Array-luokka

Määritelkäämme lopuksi lajiteltu, raja-arvot tarkistavat taulukko. Se voidaan määritellä periyttämällä sekä Array_RC- että Array_Sort-luokasta. Toteutus on tässä (jälleen kerran toteutuksemme on rajoitettu kolmeen muodostajaan ja indeksioperaattoriin; koodi on sijoitettu otsikkotiedostoon nimeltään Array_RC_S.h):

```
#ifndef ARRAY_RC_S_H
#define ARRAY_RC_S_H

#include "Array_S.C"
#include "Array_RC.C"

template <class Type>
class Array_RC_S : public Array_RC<Type>,
                   public Array_Sort<Type>
{
public:
    Array_RC_S( int sz = Array<Type>::ArraySize )
    : Array<Type>( sz )
    { clear_bit(); }

  Array_RC_S( const Array_RC_S &rca )
      : Array<Type>( rca )
      { sort( 0,Array<Type>::_size-1 ); clear_bit(); }

  Array_RC_S( const Type* arr, int sz )
      : Array<Type>( arr, sz )
      { sort( 0,Array<Type>::_size-1 ); clear_bit(); }

  Type& operator[]( int index ) {
      set_bit();
      return Array_RC<Type>::operator[]( index );
  }
};

#endif
```

Luokka perii kaksi toteutusta jokaisesta Array-luokan rajapintafunktiosta: ne Array_Sort:in ja virtuaalisen Array-kantaluokan funktiot, jotka on peritty Array_RC:n kautta (paitsi indeksioperaattori, jonka kumoava ilmentymä peritään molemmista kantaluokista). Esimerkiksi ei-virtuaalisessa johtamisessa find()-kutsu aiheuttaa moniselitteisyysvirheen — mikä peritty ilmentymä tulisi käynnistää? Virtuaalijohtamisessa annetaan kuitenkin kumoavalle Array_Sort-ilmentymäjoukolle suurempi sidontajärjestys kuin virtuaalisen kantaluokan ilmentymille, jotka on peritty Array_RC:n kautta (tätä käsitellään tarkemmin kohdassa 18.5.4). Virtuaalisessa periytymisessä tarkentamaton find()-käynnistys ratkaistaan perittyyn Array_Sort-luokan ilmentymään.

Indeksioperaattori on määritelty uudelleen sekä Array_RC- että Array_Sort-kantaluokissa ja niille annetaan samanlainen sidontajärjestys. Indeksioperaattorin tarkentamaton käynnistys Array_RC_Sort:issa on moniselitteinen. Luokalla pitää olla oma ilmentymä tai muutoin luokan käyttäjät eivät kykene käyttämään indeksioperaattoria suoraan luokan olioihin. Mitä tulkinnallisesti merkitsee käynnistää indeksioperaattori Array_RC_S-luokalle? Jotta se vaikuttaisi johdetun luokkansa lajittelun luonteeseen, sen täytyy asettaa peritty dirty_bit-tietojäsen. Jotta se vaikuttaisi johdetun luokkansa raja-arvojen tarkistuksen luonteeseen, sen täytyy tehdä testi sille annetulle indeksille. Kun se on tehty, se voi palauttaa taulukon indeksoidun elementin. Viimeiset kaksi vaihetta tekee peritty Array_RC:n indeksioperaattori. Kutsu

```
return Array_RC::operator[]( index );
```

käynnistää tämän operaattorin eksplisiittisesti. Koska se on eksplisiittinen käynnistys, virtuaalimekanismi kumoutuu. Koska se on välitön funktio, sen staattinen resoluutio johtaa koodin välittömään laajentamiseen.

Kokeilkaamme toteutusta try_array()-funktion suorituksella antamalla sille vuorollaan kokonaisluku- ja string-luokan instantiointi Array_RC_S-malliluokasta. Ohjelma on tässä:

```
#include "Array_RC_S.h"
#include "try_array.C"
#include <string>

int main()
{
   static int ia[ 10 ] = { 12,7,14,9,128,17,6,3,27,5 };
   static string sa[ 7 ] = {
               "Eeyore", "Pooh", "Tigger",
               "Piglet", "Owl", "Gopher", "Heffalump"
   };

   Array_RC_S<int> iA( ia,10 );
   Array_RC_S<string> SA( sa,7 );

   cout << "class template instantiation Array_RC_S<int>"
      << endl;
   try_array( iA );
```

```
    cout << "class template instantiation Array_RC_S<string>"
         << endl;
    try_array( SA );

    return 0;
}
```

Tässä on Array_RC_S-luokkamallin string-instantioinnin tulostus. Indeksin raja-arvon ylitys saadaan nyt siepattua.

```
class template instantiation Array_RC_S<string>

try_array: initial array values:
( 7 )< Eeyore, Gopher, Heffalump, Owl, Piglet, Pooh
     Tigger >

try_array: after assignments:
( 7 )< Eeyore, Gopher, Owl, Piglet, Pooh, Pooh
     Pooh >

try_array: memberwise initialization
( 7 )< Eeyore, Gopher, Owl, Piglet, Pooh, Pooh
     Pooh >

try_array: after memberwise copy
( 7 )< Eeyore, Piglet, Owl, Piglet, Pooh, Pooh
     Pooh >

try_array: after grow
( 11 )< <empty>, <empty>, <empty>, <empty>, Eeyore, Owl
     Piglet, Piglet, Pooh, Pooh, Pooh >

value to find: Tiggerindex returned: -1
Assertion failed: ix >= 0 && ix < size
```

Array-hierarkian esityksen tarkoitus on kuvata sekä moni- että virtuaaliperiytymisen määrittelyä ja käyttöä. Jos haluat lukea pitemmälle meneviä näkökohtia taulukkoluokan suunnittelusta, katso julkaisusta [NACKMAN94]. Useimpiin taulukkotarpeisiisi on vakiokirjaston vektoriluokka kuitenkin riittävä.

Harjoitus 18.16

Lisää Array-luokkaan lisäjäsenfunktio spy(). Jos se käynnistetään, se muistaa operaatiot, joita luokkaan on käytetty: (a) indeksikäsittelyiden lukumäärä, (b) kunkin funktion käynnistyskertojen määrä, (c) etsityn elementin arvo, kun find() on käynnistetty ja (d) onnistuneiden elementtihakujen lukumäärä. Perustele suunnitelmasi. Muokkaa koko Array-alityyppien kokoelmaa niin, että spy() toimii myös niille yhtä hyvin.

Harjoitus 18.17

Assosiatiivinen taulukko on toinen nimi vakiokirjaston kartalle (map), joka johtuu sen tuesta avainarvoon perustuvaan indeksointiin. Mitä mieltä olet, tuleeko assosiatiivisesta taulukosta hyvä ehdokas Array-luokkamme alityypitykseen? Miksi tulee tai ei tule?

Harjoitus 18.18

Toteuta uudelleen Array-hierarkiamme käyttämällä vakiokirjaston säiliöluokkamallia ja niin montaa geneeristä algoritmia kuin mahdollista.