19

Periytymisen käyttö C++:ssa

Periytymisessä voi osoitin tai viittaus kantaluokkatyyppiin viitata johdetun luokkatyypin olioon. Ohjelmamme voidaan sitten kirjoittaa käsittelemään näitä osoittimia tai viittauksia riippumattomana todellisista tyypeistä, joihin ne viittaavat. Tätä kykyä käsitellä useampaa kuin yhtä johdettua tyyppiä kantaluokan osoittimella tai viittauksella sanotaan *monimuotoisuudeksi* (*polymorphism*). Tässä luvussa katsomme kolmea kielen piirrettä, jotka tukevat monimuotoisuutta. Aluksi katsomme suorituksenaikaista tyyppitunnistusta (Run-time Type Identification; kutsutaan usein myös lyhenteellä *RTTI*), joka mahdollistaa ohjelmien hakea johdetun olion todellinen tyyppi osoittimen tai viittauksen kautta, joka osoittaa kantaluokkatyyppiin. Sitten tutkimme, kuinka poikkeusten käsittely vaikuttaa luokkaperiytymiseen: kuinka poikkeuksia voidaan määritellä luokkahierarkioina ja kuinka kantaluokkatyyppiset käsittelijät voivat hoitaa johdetun luokkatyypin poikkeuksia. Lopuksi palaamme funktion ylikuormituksen ratkaisusääntöihin nähdäksemme, kuinka luokkaperiytyminen vaikuttaa funktioargumentin mahdollisiin tyyppikonversioihin ja parhaiten elinkelpoisen funktion valintaan.

19.1 Ajonaikainen tyyppitunnistus (RTTI)

RTTI mahdollistaa ohjelmien, jotka käsittelevät olioita osoittimina tai viittauksina kantaluokkiin, hakea olioiden todelliset johdetut tyypit, joihin nämä osoittimet tai viittaukset viittaavat. C++:ssa on kaksi operaattoria RTTI:n tukemiseen:

1. `dynamic_cast`-operaattori, joka mahdollistaa ajonaikaisesti suoritettavat tyyppikonversiot ja ohjelmien siirtyilyn luokkahierarkiassa turvallisesti, konvertoimalla osoittimen kantaluokkaan osoittimeksi johdettuun luokkaan tai konvertoimalla kantaluokkaan viittaavan *lvaluen* viittaukseksi johdettuun luokkaan vain, kun konversion on todella taattu onnistuvan.
2. `typeid`-operaattori, joka ilmaisee olion todellisen johdetun tyypin, johon osoitin tai viittaus viittaa.

Kuitenkin, jotta johdetun luokkatyypin tieto voitaisiin hakea, joko dynamic_cast- tai typeid-operaattorin operandin pitää olla luokkatyyppi, jossa on yksi tai useampi virtuaalifunktio. Tämä tarkoittaa, että RTTI-operaattorit ovat ajonaikaisia tapahtumia luokille, joilla on virtuaalifunktioita, ja käännöksenaikaisia tapahtumia kaikille muille tyypeille. Tässä kohdassa katsomme hieman tarkemmin tukea, jota näille kahdelle operaattorille on tarjolla.

RTTI:n käyttö ohjelmissamme on joskus tarpeen, kun toteutamme sovelluksia kuten virheenetsintäohjelmat (debuggerit) tai tietokannat, joissa käsiteltävien olioiden tyyppi tiedetään vasta suorituksen aikana. Se saadaan selville tutkimalla RTTI-tietoa, joka on tallennettu olioiden tyyppeihin. RTTI-operaattoreiden käyttöä tulisi kuitenkin minimoida. C++:n staattista tyyppijärjestelmää (kääntäjän tyyppitarkistusta) tulisi käyttää aina, kun se on mahdollista, koska se on turvallisempi ja tehokkaampi.

19.1.1 Operaattori dynamic_cast

Operaattoria dynamic_cast voidaan käyttää luokkatyyppiseen olioon viittaavan osoittimen konvertoimiseen osoittimeksi toiseen luokkaan samassa luokkahierarkiassa. Operaattoria dynamic_cast voidaan käyttää myös, kun konvertoidaan luokkatyyppisen olion lvalue viittaukseksi luokkaan samassa luokkahierarkiassa. Toisin kuin muut C++:n tukemat tyyppimuunnokset, dynamic_cast tehdään suorituksenaikaisesti. Ellei osoitinta tai lvalueta pysty muuntamaan konversion kohdetyypiksi, dynaaminen tyyppimuunnos epäonnistuu. Jos dynaaminen tyyppimuunnos osoitintyypiksi epäonnistuu, dynamic_cast:in arvo on 0. Jos dynaaminen tyyppimuunnos viittaustyypiksi epäonnistuu, heitetään poikkeus. Näytämme esimerkkejä dynamic_cast-operaatioiden epäonnistumisista myöhemmin.

Ennen kuin tutkimme dynamic_cast-operaattorin toimintaa tarkemmin, tutkikaamme, miksi käyttäjä voisi tarvita dynamic_cast-operaattoria C++-ohjelmassa. Oletetaan, että ohjelmamme käyttää luokkakirjastoa esittääkseen erilaisia työntekijöitä yrityksessämme. Hierarkian luokat tukevat jäsenfunktioita, joilla lasketaan yrityksemme palkkalista. Esimerkiksi:

```
class employee {
public:
    virtual int salary();
};

class manager : public employee {
public:
    int salary();
};

class programmer : public employee {
public:
    int salary();
};
```

```
void company::payroll( employee *pe ) {
    // käyttö: pe->salary()
}
```

Yrityksessämme on erilaisia työntekijöitä. Yrityksen (company) payroll()-jäsenfunktion parametri on osoitin työntekijäluokkaan, joka voi viitata joko johtajatyyppiin (manager) tai ohjelmoijatyyppiin (programmer). Koska payroll() kutsuu virtuaalista salary()-jäsenfunktiota, kutsutaan joko manager- tai programmer-luokassa vastaavaa sen kumoavaa funktiota riippuen työntekijästä, johon pe viittaa.

Olettakaamme, että employee-luokka ei sovi tarpeisiimme enää ja haluamme muokata sitä. Haluamme lisätä jäsenfunktion nimeltään bonus() käytettäväksi salary()-jäsenfunktion kanssa, kun lasketaan yrityksemme palkkalistaa. Voimme tehdä sen lisäämällä virtuaalisen jäsenfunktion employee-hierarkian luokkiin. Esimerkiksi:

```
class employee {
public:
    virtual int salary();
    virtual int bonus();
};

class manager : public employee {
public:
    int salary();
};

class programmer : public employee {
public:
    int salary();
    int bonus();
};

void company::payroll( employee *pe )
{
    // käyttö: pe->salary() ja pe->bonus()
}
```

Jos payroll():in pe-parametri viittaa manager-tyyppiseen olioon, kutsutaan virtuaalista bonus()-jäsenfunktiota, joka on määritelty employee-kantaluokkaan, koska manager-luokka ei kumoa employee-luokkaan määriteltyä bonus()-virtuaalifunktiota. Jos payroll():in pe-parametri viittaa programmer-tyyppiseen olioon, kutsutaan virtuaalista bonus()-jäsenfunktiota, joka on määritelty programmer-luokkaan.

Kun virtuaalifunktioita lisätään luokkahierarkiaan, tulee välttämättömäksi kääntää uudelleen kaikki luokkahierarkian luokkajäsenfunktiot. Voimme lisätä virtuaalisen bonus()-jäsenfunktion, jos meillä on pääsy lähdekoodiin, jolla toteutetaan employee-, manager- ja programmer-luokkien jäsenfunktiot. Näin ei aina ole asianlaita. Jos edellisen luokkahierarkian on tehnyt kolmannen osapuolen ohjelmakirjaston valmistaja, käytettävissämme on vain

otsikkotiedostot, joissa kuvataan kirjaston luokkien rajapinnat ja objektitiedostot. Luokkien jäsenfunktioiden lähdekoodi ei ehkä ole saatavilla. Siinä tapauksessa ei luokkien jäsenfunktioiden uudelleen kääntäminen ole mahdollista.

Jos haluamme laajentaa luokkakirjastoa, emme voi lisätä virtuaalisia jäsenfunktioita. Voimme silti haluta lisätä toiminnallisuutta, jolloin dynamic_cast-operaattorista tulee tarpeellinen.

Operaattoria dynamic_cast käytetään, jotta saataisiin osoitin johdettuun luokkaan jonkin sen yksityiskohdan käyttämiseksi, joka ei muutoin olisi käytettävissä. Oletetaan esimerkiksi, että laajennamme kirjastoa lisäämällä bonus()-jäsenfunktion programmer-luokkaan. Voimme lisätä tämän jäsenfunktion programmer-luokan määrittelyyn, joka on saatavilla otsikkotiedostossa, ja määritellä tämän uuden jäsenfunktion johonkin omaan ohjelmatekstitiedostoon:

```
class employee {
public:
    virtual int salary();
};

class manager : public employee {
public:
    int salary();
};

class programmer : public employee {
public:
    int salary();
    int bonus();
};
```

Muista, että payroll()-funktiomme vastaanottaa parametrin, joka on osoitin employee-kantaluokkaan. Voimme käyttää dynamic_cast-operaattoria saadaksemme osoittimen johdettuun programmer-luokkaan ja käyttää tätä osoitinta bonus()-jäsenfunktion kutsumiseen kuten seuraavassa:

```
void company::payroll( employee *pe )
{
    programmer *pm = dynamic_cast< programmer* >( pe );

    // jos pe viittaa programmer-tyyppiseen olioon,
    // dynamic_cast on onnistunut
    // ja pm viittaa programmer-olion alkuun
    if ( pm ) {
        // käytä pm-osoitinta kutsuaksesi programmer::bonus()
    }
```

```
    // jos pe ei viittaa programmer-tyyppiseen olioon,
    // dynamic_cast epäonnistuu
    // ja pm saa arvon 0
    else {
        // employee:n jäsenfunktioiden käyttö
    }
}
```

Dynaaminen tyyppimuunnos

```
dynamic_cast< programmer* >( pe )
```

konvertoi pe-operandinsa programmer*-tyypiksi. Tyyppimuunnos onnistuu, jos pe viittaa programmer-tyyppiseen olioon, muussa tapauksessa tyyppimuunnos epäonnistuu ja dynamic_cast saa arvon 0.

Tästä syystä dynamic_cast-operaattori tekee kaksi operaatiota kerralla. Se varmistuu, että pyydetty tyyppimuunnos on todella kelvollinen ja sitten vasta, kun tyyppimuunnos on kelvollinen, se tekee sen. Varmistuminen tapahtuu suorituksen aikaan. dynamic_cast on turvallisempi kuin muut C++:n tyyppimuunnosoperaatiot, koska muut eivät varmistu, että tyyppimuunnos voidaan todella tehdä.

Jos edellisessä esimerkissä pe todella osoittaa suorituksen aikana programmer-olioon, dynamic_cast-operaatio onnistuu ja pm alustetaan osoittamaan programmer-olioon. Muussa tapauksessa dynamic_cast-operaation arvo on 0 ja pm alustetaan arvolla 0. Tarkistamalla pm:n arvon funktio company::payroll() tietää, milloin pe viittaa programmer-olioon. Se voi sitten käyttää programmer::bonus()-jäsenfunktiota laskeakseen ohjelmoijan palkan. Jos dynamic_cast epäonnistuu siitä syystä, että pe viittaa manager-olioon, käytetään sen sijaan yleisempää palkanlaskentaa, joka ei käytä hyväkseen uutta programmer::bonus()-jäsenfunktiota.

dynamic_cast-operaattoria on käytetty turvalliseen tyyppimuunnokseen osoittimesta kantaluokkaan osoittimeksi johdettuun luokkaan, jota usein kutsutaan termillä turvallinen *tyyppimuunnos alaspäin* (*downcasting*). Sitä käytetään, kun täytyy käyttää johdetun luokan piirteitä, joita ei ole kantaluokassa. Johdetun luokkatyypin olioiden käsittely osoittimella kantaluokkatyyppiin hoidetaan usein automaattisesti virtuaalifunktioiden kautta. Kuitenkin joissakin tapauksissa virtuaalifunktioiden käyttö ei ole mahdollista. dynamic_cast antaa vaihtoehtoisen mekanismin sellaisiin tilanteisiin, vaikka tämä mekanismi on virhealttiimpaa kuin virtuaaliset jäsenfunktiot ja sitä tulisikin käyttää huolella.

Eräs mahdollinen virhe dynamic_cast-operaattorin käytössä on, että ei kunnolla testata sen tulosta, eli onko sen arvo 0. Jos se on, ei tulosta voi silloin käyttää aivan kuin se viittaisi luokkaolioon. Esimerkiksi:

```
void company::payroll( employee *pe ) {
    programmer *pm = dynamic_cast< programmer* >( pe );

    // potentiaalinen virhe: käyttää pm-osoitinta ennen kuin testaa sen arvon
    static int variablePay = 0;
    variablePay += pm->bonus();
```

```
    // ...
}
```

dynamic_cast-operaattorin tulos pitää aina testata, jotta varmistutaan tyyppimuunnoksen onnistumisesta ennen sen tuloksena syntyvän osoittimen käyttöä. company::payroll()-funktion parempi määrittely on seuraavassa:

```
void company::payroll( employee *pe )
{
    // dynamic_cast ja testaus ehtolausekkeessa
    if ( programmer *pm = dynamic_cast< programmer* >( pe ) ) {
        // käytä pm-osoitinta programmer::bonus()-kutsuun
    }
    else {
        // käytä employee:n jäsenfunktioita
    }
}
```

dynamic_cast-operaation tulosta käytetään pm-muuttujan alustamiseen if-lauseen ehtolausekkeessa. Tämä on mahdollista, koska esittelyt ehtolausekkeissa johtavat arvoihin. If-lauseen tosiarvon mukainen haara suoritetaan, jos pm ei ole nolla; tarkoittaa, jos dynamic_cast onnistuu, koska pe-osoitin todella osoittaa programmer-olioon, muussa tapauksessa esittely johtaa arvoon 0, jolloin suoritetaan else-haara. Koska dynamic_cast-operaatio ja sen tuloksen testaus ovat nyt yhdessä lauseessa, ei ole mahdollista lisätä koodia erehdyksessä niiden väliin ja käyttää pm-osoitinta ennen sen asianmukaista testaamista.

Edellisessä esimerkissä dynamic_cast-operaatio konvertoi osoittimen kantaluokkaan osoittimeksi johdettuun luokkaan. dynamic_cast-operaatiota voidaan käyttää myös konvertoitaessa kantaluokkatyyppinen lvalue johdetun luokan tyyppiseksi viittaukseksi. Sellaisen dynamic_cast-operaation syntaksi on seuraava

```
dynamic_cast< Type& >( lval )
```

jossa Type& on konversion kohdetyyppi ja lval on kantaluokkatyypin lvalue. dynamic_cast-operaatio konvertoi lval-operandin halutuksi Type&-tyypiksi vain, jos lval todella viittaa olioon, jolla on kantaluokka tai johdettu luokka, joka on tyyppiä Type.

Koska ei ole olemassa null-viittausta (katso kohta 3.6), ei ole mahdollista varmistaa viittauksen onnistumista dynamic_cast:in yhteydessä vertaamalla sen tulosta (viittaus, joka on dynamic_cast:in tuloksena) arvoon 0. Jos edellisessä esimerkissä halutaan käyttää viittauksia osoittimien sijasta, ehtoa

```
if ( programmer *pm = dynamic_cast< programmer* >( pe ) )
```

ei voi kirjoittaa kuten seuraavassa:

```
if ( programmer &pm = dynamic_cast< programmer& >( pe ) )
```

Virhetilanne raportoidaan eri tavalla dynamic_cast-operaattorilla, kun sitä käytetään viittaustyypin konvertointiin. Viittauksen dynaaminen tyyppimuunnos (dynamic_cast), joka epäonnistuu, heittää poikkeuksen.

Edellinen esimerkki pitää kirjoittaa uudelleen käyttämään dynamic_cast-operaattoria viittaustyyppiin seuraavasti:

```
#include <typeinfo>
void company::payroll( employee &re )
{
    try {
        programmer &rm = dynamic_cast< programmer & >( re );
        // käytä re:tä, kun kutsut programmer::bonus()-funktiota
    }
    catch ( std::bad_cast ) {
        // käytä employee:n jäsenfunktioita
    }
}
```

Jos viittauksen dynaaminen tyyppimuunnos epäonnistuu, heitetään bad_cast-tyyppinen poikkeus. Luokkatyyppi bad_cast on määritelty C++-vakiokirjastoon. Jotta voimme viitata tähän tyyppiin ohjelmassamme kuten edellisessä esimerkissä, pitää ottaa mukaan otsikkotiedosto <typeinfo>. (Katsomme C++-vakiokirjastoon määriteltyjä poikkeuksia seuraavassa kohdassa.)

Milloin tulisi viittauksen dynaamista tyyppimuunnosta käyttää osoittimen dynaamisen tyyppimuunnoksen sijasta? Tämä on todella ohjelmoijan valinta. Viittauksen dynaamisessa tyyppimuunnoksessa ei ole mahdollista olla huomioimatta sen epäonnistumista ja käyttää tulosta testaamatta sitä asianmukaisesti, kuten voidaan tehdä osoittimen dynaamisessa tyyppimuunnoksessa. Poikkeusten käyttö lisää kuitenkin vastaavasti ohjelman suoritusaikaa (kuten kerrottiin luvussa 11) ja saa jotkut ohjelmoijat pitämään osoittimien dynaamisia tyyppimuunnoksia parempina.

19.1.2 Operaattori typeid

Toinen operaattori RTTI-tukea varten on typeid-operaattori. typeid-operaattori mahdollistaa ohjelman kysyä lausekkeelta: minkä tyyppinen olet? Jos lauseke on luokkatyyppinen ja luokka sisältää yhden tai useamman virtuaalisen jäsenfunktion, silloin vastaus voi olla erilainen kuin itse lausekkeen tyyppi. Jos esimerkiksi lauseke on viittaus kantaluokkaan, typeid-operaattori ilmaisee taustalla olevan olion johdetun luokan tyypin. Esimerkiksi:

```
#include <typeinfo>

programmer pobj;
employee &re = pobj;

// katsomme name()-funktiota myöhemmin type_info-alikohdassa
// name() palauttaa C-tyylisen merkkijonon: "programmer"
cout << typeid( re ).name() << endl;
```

typeid-operaattorin operandi on employee-tyyppi. Koska kuitenkin re on viittaus luokkatyyppiin, jolla on virtuaalifunktioita, typeid-operaattorin tulos ilmaisee taustalla olevan olion programmer-tyyppiseksi (eikä employee-tyypiksi, joka on re-operandin tyyppi). Kun typeid-operaattoria käytetään, pitää ohjelmatekstitiedostoon ottaa mukaan otsikkotiedosto <typeinfo>, joka on määritelty C++-vakiokirjastoon kuten on tehty tässä esimerkissä.

Mihin typeid-operaattoria käytetään? Sitä käytetään edistyneissä järjestelmäohjelmointiympäristöissä, kun rakennetaan esimerkiksi virheenetsintäohjelmia (debuggereita) tai kun haetaan olioita tietokannasta. Kun sellaisissa järjestelmissä ohjelma käsittelee oliota osoittimen tai viittauksen kautta kantaluokkaan, pitää ohjelman selvittää käsiteltävän olion todellinen tyyppi, jotta sen ominaisuudet voidaan listata virheenetsintäistunnon aikana tai tallentaa ja hakea olio asianmukaisesti tietokannasta. Jotta olion todellinen tyyppi voidaan selvittää, voidaan käyttää typeid-operaattoria.

typeid-operaattoria voidaan käyttää minkä tahansa tyyppisten lausekkeiden ja tyyppinimien kanssa. Esimerkiksi sisäisten tyyppien lauseketta kuten myös vakioita voidaan käyttää typeid-operaattorin operandina. Kun operandi ei ole luokkatyyppi, silloin typeid-operaattori ilmaisee operandin tyypin:

```
int iobj;

cout << typeid( iobj ).name() << endl; // tulostaa: int
cout << typeid( 8.16 ).name() << endl; // tulostaa: double
```

Kun typeid-operaattorin operandi on luokkatyyppi, mutta ei sellaisen luokan, jossa on virtuaalifunktioita, myös silloin typeid-operaattori ilmaisee operandin tyypin eikä taustalla olevan olion tyyppiä:

```
class Base { /* ei virtuaalifunktiota */ };
class Derived : public Base { /* ei virtuaalifunktiota */ };

Derived dobj;
Base *pb = &dobj;

cout << typeid( *pb ).name() << endl; // tulostaa: Base
```

typeid-operaattorin tyyppi on Base, joka on *pb-lausekkeen tyyppi. Koska Base-luokassa ei ole virtuaalifunktioita, typeid-operaattorin tulos ilmaisee, että lausekkeen tyyppi on Base, vaikka taustalla olevan olion tyyppi, johon pb viittaa, on Derived.

typeid-operaattorin tulosta voidaan vertailla. Esimerkiksi:

```
#include <typeinfo>

employee *pe = new manager;
employee& re = *pe;

if ( typeid( pe ) == typeid( employee* ) ) // tosi
    // tee jotain
/*
```

```
if ( typeid( pe ) == typeid( manager* ) )  // epätosi
if ( typeid( pe ) == typeid( employee ) )  // epätosi
if ( typeid( pe ) == typeid( manager ) )  // epätosi
*/
```

If-lauseen ehdossa vertaillaan typeid-operaattorin käyttöä operandin kanssa, joka on lausekkeen tulos. Siinä on käytetty typeid-operaattoria operandin kanssa, joka on tyyppinimi. Huomaa, että vertailu

```
typeid( pe ) == typeid( employee* )
```

johtaa tosi-arvoon. Tämä voi näyttää yllättävältä käyttäjille, jotka ovat tottuneet kirjoittamaan:

```
// kutsuu virtuaalifunktiota
pe->salary();
```

joka johtaa johdetun manager-luokan salary()-funktion kutsuun. typeid(pe) käyttäytyy eri tavalla kuin virtuaalifunktioiden kutsumekanismi. Tämä siksi, koska pe-operandi on osoitin eikä luokkatyyppi. Kun haetaan johdettua luokkatyyppiä, pitää typeid-operaattorin operandin olla luokkatyyppi (ja sisältää virtuaalifunktioita). Lauseke typeid(pe) ilmaisee pe:n tyypin: osoitin employee:hin. Se on yhtäsuuri vertailulausekkeen typeid(employee*) kanssa, kun taas johtaa muissa vertailuissa epätosi-arvoon.

Kun lauseketta *pe käytetään typeid:in kanssa, sen tulos ilmaisee taustalla olevan olion tyypin, johon pe osoittaa:

```
typeid( *pe ) == typeid( manager )  // tosi
typeid( *pe ) == typeid( employee ) // epätosi
```

Koska näissä vertailuissa *pe on luokkatyyppinen lauseke ja koska luokassa on virtuaalifunktioita, typeid:n tulos ilmaisee taustalla olevan olion tyypin, johon operandi viittaa, eli manager-tyypin.

typeid-operaattoria voidaan käyttää myös viittausten kanssa. Esimerkiksi:

```
typeid( re ) == typeid( manager )    // tosi
typeid( re ) == typeid( employee )   // epätosi

typeid( &re ) == typeid( employee* ) // tosi
typeid( &re ) == typeid( manager* )  // epätosi
```

Kahdessa ensimmäisessä vertailussa operandi re on luokkatyyppinen virtuaalifunktioineen. typeid-operaattorin tulos ilmaisee taustalla olevan olion tyypin, johon re viittaa. Kahdessa viimeisessä vertailussa operandi &re on osoitintyyppi. typeid-operaattorin tulos ilmaisee operandin tyypin: employee*.

Itse asiassa typeid-operaattori palauttaa luokkaolion, jonka tyyppi on type_info. Luokkatyyppi type_info on määritelty otsikkotiedostoon <typeinfo>. Luokan rajapinnassa kuvataan, mitä typeid-operaattorin tuloksella voidaan tehdä. Tutkimme tätä rajapintaa seuraavassa alikohdassa.

19.1.3 Type_info-luokka

Type_info-luokan täsmällinen määrittely on toteutusriippuvainen, mutta tietyt tämän luokan piirteet ovat samoja jokaisessa C++-ohjelmassa:

```
class type_info {
    // toteutusriippuvainen esitystapa
private:
    type_info( const type_info& );
    type_info& operator= ( const type_info& );
public:
    virtual ~type_info();

    int operator==( const type_info& ) const;
    int operator!=( const type_info& ) const;

    const char * name() const;
};
```

Koska type_info-luokan kopiointimuodostaja ja kopioinnin sijoitusoperaattori ovat yksityisiä jäseniä, käyttäjät eivät voi määritellä type_info-tyyppisiä olioita ohjelmiinsa. Esimerkiksi:

```
#include <typeinfo>

type_info t1; // virhe: ei oletusmuodostajaa
        // virhe: yksityinen kopiointimuodostaja
type_info t2 ( typeid( unsigned int ) );
```

Ainoa tapa luoda type_info-olioita ohjelmaan on käyttää typeid-operaattoria.

Luokassa on myös ylikuormitettuja vertailuoperaattoreita. Nämä operaattorit mahdollistavat, että voidaan vertailla kahta type_info-oliota ja sitten vertailla typeid-operaattorilla saatuja tuloksia, kuten olemme nähneet edellisessä alikohdassa.

```
typeid( re )  == typeid( manager )    // tosi
typeid( *pe ) != typeid( employee )   // epätosi
```

Funktio name() palauttaa C-tyylisen merkkijonon type_info-olion edustaman tyypin nimelle. Tätä funktiota voidaan käyttää ohjelmissamme seuraavasti:

```
#include <typeinfo>
int main() {
    employee *pe = new manager;

    // tulostaa: "manager"
    cout << typeid( *pe ).name() << endl;
}
```

Jotta name()-jäsenfunktiota voidaan käyttää, ei saa unohtaa ottaa mukaan <typeinfo>-otsikkotiedostoa.

Type_info:n name()-jäsenfunktion kautta saatu tyyppinimi on ainoa taattu tieto, joka saadan

kaikissa C++-toteutuksissa. Kuten tämän kohdan alussa mainittiin, RTTI-tuki on toteutusriippuvainen ja joissakin toteutuksissa on type_info-luokalle lisäjäsenfunktioita, joita ei ole lueteltu tässä. Sinun tulisi katsoa kääntäjäsi hakukäsikirjasta, millaista RTTI-tukea se voi antaa. Millaista lisätukea voisi olla? Periaatteessa mitä tahansa tietoa, jota kääntäjä voi antaa tyypistä, voidaan lisätä. Esimerkiksi:

1. Luettelo luokan jäsenfunktioista.
2. Millaiselta tämän luokkatyypin olion sommittelu näyttää muistissa: kuinka jäsen- ja kanta-alioliot on yhdistetty toisiinsa.

Eräs yleinen tekniikka, joka voidaan valita toteutukseen RTTI-tuen laajentamiseksi, on lisätä tietoja luokkatyypistä, joka on johdettu type_info-luokasta. Koska type_info-luokka sisältää virtuaalituhoajan, voidaan dynamic_cast-operaattoria käyttää pääteltäessä, onko tietyntyyppistä, laajennettua RTTI-tukea saatavilla. Sanokaamme esimerkiksi, että toteutuksessa on lisätukea saatavilla RTTI:lle extended_type_info-nimisen luokan kautta, joka on johdettu type_info-luokasta. Käyttämällä dynaamista tyyppimuunnosta (dynamic_cast), ohjelma voi saada selville, onko typeid-operaattorin palauttama type_info-olio extended_type_info-tyyppi vai voidaanko RTTI-lisätukea saada ohjelmaan kuten seuraavassa:

```
#include <typeinfo>

// typeinfo sisältää määrittelyn extended_type_info:lle

typedef extended_type_info eti;

void func( employee* p )
{
    // alaspäintyypitys: type_info* ----> extended_type_info*
    if ( eti *eti_p = dynamic_cast<eti *>( &typeid( *p ) ) )
    {
        // jos dynaaminen tyyppimuunnos onnistuu,
        // käytä extended_type_info-tietoa eti_p:n kautta
    }
    else
    {
        // jos dynaaminen tyyppimuunnos epäonnistuu,
        // käytä type_info:n vakiotietoa
    }
}
```

Jos dynaaminen tyyppimuunnos onnistuu, typeid-operaattori palauttaa extented_type_info-tyyppisen olion, joka tarkoittaa, että toteutuksesta on saatavilla RTTI-lisätukea ohjelman käyttöön. Jos dynaaminen tyyppimuunnos epäonnistuu, on silloin vain RTTI-perustukea saatavilla ohjelman käyttöön.

Harjoitus 19.1

Olkoon seuraava luokkahierarkia, jossa jokaisessa luokassa on määritelty oletusmuodostaja ja virtuaalituhoaja:

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

Mitkä seuraavista dynaamisista tyyppimuunnoksista epäonnistuvat, vai epäonnistuuko yksikään?

```
(a) D *pd = new D;
    A *pa = dynamic_cast< A* >( pd );

(b) A *pa = new C;
    C *pc = dynamic_cast< C* >( pa );

(c) B *pb = new B;
    D *pd = dynamic_cast< D* >( pb );

(d) A *pa = new D;
    X *px = dynamic_cast< X* >( pa );
```

Harjoitus 19.2

Kerro, milloin käyttäisit dynaamista tyyppimuunnosta (dynamic_cast) virtuaalifunktion sijasta?

Harjoitus 19.3

Käytä harjoituksen 19.1 luokkahierarkiaa ja kirjoita uudelleen seuraava koodikatkelma niin, että siinä käytetään viittaukselle dynaamista tyyppimuunnosta konvertoitaessa lauseke *pa tyypiksi D&:

```
if ( D *pd = dynamic_cast< D* >( pa ) )
     // käytä D:n jäseniä
}
else {
     // käytä A:n jäseniä
}
```

Harjoitus 19.4

Olkoon seuraava luokkahierarkia, jossa jokaiseen luokkaan on määritelty oletusmuodostaja ja virtuaalituhoaja:

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

Mikä tyyppinimi tulostetaan vakiovirtaan kussakin tapauksessa?

```
(a) A *pa = new D;
    cout << typeid( pa ).name() << endl;

(b) X *px = new D;
    cout << typeid( *px ).name() << endl;

(c) C cobj;
    A& ra = cobj;
    cout << typeid( &ra ).name() << endl;

(d) X *px = new D;
    A& ra = *px;
    cout << typeid( ra ).name() << endl;
```

19.2 Poikkeukset periytymisessä

Poikkeusten käsittely tarjoaa vakion, kielitasoisen piirteen vastata ohjelman suorituksenaikaisiin poikkeavuuksiin. C++ tukee yhtenäistä syntaksia ja tyyliä poikkeusten käsittelyyn ja myös ohjelmoijien tekemien poikkeusten käsittelyiden hienosäätöä. Perus-C++:n poikkeusten käsittelyn tuki on esitelty luvussa 11. Luvussa 11 näytetään, kuinka ohjelma voi heittää poikkeuksen; kun poikkeus on heitetty, miten ohjelman kulku siirtyy poikkeuksen käsittelijälle, jos sellainen on olemassa poikkeukselle; ja kuinka poikkeuksen käsittelijä liittyy try-lohkoihin.

Poikkeusten käsittelyn mahdollisuuksista tulee vieläkin erilaisempaa, kun luokkatyyppien hierarkiaa käytetään poikkeuksina. Tässä kohdassa tutkimme, kuinka kirjoitamme ohjelmamme heittämään ja käsittelemään poikkeuksia sellaisista hierarkioista.

19.2.1 Luokkahierarkioiksi määritellyt poikkeukset

Luvussa 11 käytimme kahta luokkatyyppiä kuvaamaan eri poikkeuksia, joita iStack-luokkamme jäsenfunktiot heittivät:

```
class popOnEmpty { ... };
class pushOnFull { ... };
```

Käytännön C++-ohjelmissa poikkeuksia edustavat luokkatyypit on useimmiten järjestetty ryhmiin eli hierarkioihin. Miltä voisi näyttää kahden poikkeusluokkamme muu poikkeushierarkia?

Voimme määritellä kantaluokan nimeltään Excp, josta kumpikin poikkeusluokka on johdettu. Tämä kantaluokka kapseloi molemmille johdetuille luokille yhteiset tietojäsenet ja jäsenfunktiot:

```
class Excp { ... };
class popOnEmpty : public Excp { ... };
class pushOnFull : public Excp { ... };
```

Eräs operaatio, jonka Excp-kantaluokka voi tehdä, on virheilmoitusten tulostus. Tämä piirre on molempien poikkeusluokkien käytössä hierarkiassa:

```
class Excp {
public:
    // tulost virheilmoitus
    static void print( string msg ) {
        cerr << msg << endl;
    }
};
```

Poikkeusluokkahierarkiaa voidaan määritellä edelleen uudelleen. Excp-kantaluokasta voidaan johtaa muita luokkia, jotka kuvaavat tarkemmin mahdollisia poikkeuksia, joita ohjelmissamme havaitaan:

```
class Excp { ... };

class stackExcp : public Excp { ... };
    class popOnEmpty : public stackExcp { ... };
    class pushOnFull : public stackExcp { ... };

class mathExcp : public Excp { ... };
    class zeroOp : public mathExcp { ... };
    class divideByZero : public mathExcp { ... };
```

Nämä lisähienoudet mahdollistavat tarkemman tunnistuksen ohjelman poikkeavuuksille, joita voi ohjelmassamme tapahtua. Poikkeusten lisäluokat on järjestetty kerroksittain. Kun hierarkiasta tulee syvempi, tulee jokaisesta kerroksesta erityisempi poikkeus. Esimerkiksi ensimmäistä ja kaikkein yleisintä kerrosta edellisessä poikkeusluokkahierarkiassa edustaa Excp-luokka. Toinen kerros erikoistaa Excp-luokan kahteen eri luokkaan: stackExcp (poikkeuksille, jotka tapahtuva iStack-luokkaa käsiteltäessä) ja mathExcp (poikkeuksille, jotka tapahtuvat math-kirjastomme funktioissa). Kolmas ja kaikkein erikoistunein hierarkiamme kerros jalostaa poikkeusluokkia vielä edelleen. Luokat popOnEmpty ja pushOnFull määrittelevät kaksi erilaista stackExcp-poikkeusta, kun taas luokat zeroOp- ja divideByZero määrittelevät kaksi eri mathExcp-poikkeusta.

Katsomme seuraavissa alikohdissa, kuinka heittää ja käsitellä poikkeustyyppejä, jotka ovat

luokkia juuri määrittelemässämme hierarkiassa.

19.2.2 Luokkatyyppisen poikkeuksen heittäminen

Nyt, kun olemme nähneet luokkatyyppejä tarkemmin, tutkikaamme, mitä tapahtuu, kun iStack-jäsenfunktio push() heittää poikkeuksen:

```
void iStack::push( int value )
{
    if ( full() )
        // arvo tallennettu poikkeusolioon
        throw pushOnFull( value );
    // ...
}
```

Monet vaiheet tapahtuvat tämän throw-lausekkeen suorituksen seurauksena:

1. Throw-lauseke luo tilapäisen pushOnFull-luokkatyyppisen olion kutsumalla luokan muodostajaa.
2. Luodaan pushOnFull-tyyppinen poikkeusolio, jotta se voidaan välittää poikkeuksen käsittelijälle. Poikkeusolio on kohdassa 1 throw-lausekkeella luodun tilapäisolion kopio. Se luodaan kutsumalla pushOnFull-luokan kopiointimuodostajaa.
3. Tilapäinen olio, joka luotiin kohdassa 1 throw-lausekkeella, tuhotaan ennen kuin käsittelijän etsintä alkaa.

Voit nyt ihmetellä, miksi vaihetta 2 tarvitaan — miksi poikkeusolio luotiin? Throw-lauseke

```
pushOnFull( value );
```

luo tilapäisen olion, joka tuhotaan throw-lausekkeen lopussa. Poikkeuksen pitää kuitenkin kestää siihen saakka, kunnes on löytynyt käsittelijä, joka voi olla monta funktiota ylempänä funktiokutsuketjussa. Siksi on tarpeen kopioida tilapäinen olio muistipaikkaan, jota kutsutaan *poikkeusolioksi* ja jonka taataan kestävän siihen saakka, kunnes poikkeus on käsitelty. Joissakin tapauksissa on mahdollista, että toteutus luo poikkeusolion suoraan luomatta vaiheen 1 tilapäistä oliota. Tämän tilapäisen olion eliminointia ei vaadita eikä se aina ole mahdollistakaan.

Koska poikkeusolio on luotu kopioimalla throw-lausekkeen arvo, on heitetyllä poikkeuksella aina täsmälleen lausekkeen tyyppi, joka on määritetty throw-lausekkeessa. Esimerkiksi:

```
void iStack::push( int value ) {
    if ( full() ) {
        pushOnFull except( value );
        stackExcp *pse = &except;
        throw *pse; // poikkeusolion tyyppi on stackExcp
    }
    // ...
}
```

Lausekkeen *pse tyyppi on stackExcp. Luodun poikkeusolion tyyppi on stackExcp, vaikka pse viittaa olioon, jonka todellinen tyyppi on pushOnFull. Olion, johon throw-lauseke viittaa, todellista tyyppiä ei koskaan tutkita poikkeusoliota luotaessa. Poikkeusta ei siksi käsitellä catch-lauseen tyypillä pushOnFull.

Throw-lausekkeen toimenpiteet tuovat esille joitakin rajoituksia luokkiin, joita voidaan käyttää poikkeusolioiden luomiseen. Throw-lauseke iStack:in push()-jäsenfunktiossa on virheellinen, jos

1. Luokalla pushOnFull ei ole muodostajaa, joka hyväksyy int-tyyppisen argumentin tai jos tätä muodostajaa ei päästä käsittelemään.
2. Luokalla pushOnFull on joko kopiointimuodostaja tai tuhoaja, joka ei ole käsiteltävissä.
3. Luokka pushOnFull on abstrakti kantaluokka, koska ohjelma ei voi luoda abstraktin luokkatyypin oliota (kuten mainittiin kohdassa 17.1)

19.2.3 Luokkatyyppisen poikkeuksen käsittely

Kun poikkeukset on järjestetty luokkahierarkioihin, luokkatyyppinen poikkeus voidaan siepata catch-lauseella tuon luokkatyypin julkisessa kantaluokassa. Esimerkiksi pushOnFull-tyyppinen poikkeus voidaan käsitellä catch-lausessa, joka on tarkoitettu poikkeustyypeille stackExcp tai Excp.

```
int main( ) {
    try {
        // ...
    }
    catch ( Excp ) {
        // käsittelee popOnEmpy- ja pushOnFull-poikkeukset
    }
    catch ( pushOnFull ) {
        // käsittelee pushOnFull-poikkeukset
    }
}
```

Catch-lauseiden järjestys edellisessä esimerkissä ei ole kaikkein parhain. Huomaatko, miksi? Muista, että catch-lauseet tutkitaan siinä järjestyksessä, jossa ne esiintyvät try-lohkon jälkeen. Kun catch-lause on löytynyt poikkeukselle, ei catch-lauseita tutkita enää edelleen. Koska edellisessä esimerkissä Excp-poikkeusten catch-lause käsittelee pushOnFull-tyyppiset poikkeukset, ei pushOnFull-poikkeuksille erikoistettuun catch-lauseeseen koskaan saavuta! Catch-lauseiden asianmukainen järjestys on seuraava:

```
catch ( pushOnFull ) {
    // käsittelee pushOnFull-poikkeukset
}
catch ( Excp ) {
    // käsittelee muut poikkeukset
}
```

Johdetun luokkatyypin catch-lauseen pitää esiintyä ensimmäisenä. Tämä varmistaa sen, että kantaluokan catch-lauseeseen saavutaan vain, jos muut catch-lauseet eivät käy.

Kun poikkeukset on järjestetty luokkahierarkioihin, luokkakirjaston käyttäjät voivat valita käsittelytason, jolla he voivat käsitellä kirjastosta heitettyjä poikkeuksia. Kun esimerkiksi main()-funktiota kirjoittaessamme päätimme, että sovelluksemme käsittelee poikkeustyypin pushOnFull jollakin erityisellä tavalla, niin siksi teimme erikoistetun catch-lauseen tämänkaltaiselle poikkeukselle. Päätimme myös, että sovelluksemme käsittelisi myös kaikki muut poikkeukset yleisemmällä tavalla. Esimerkiksi:

```
catch ( pushOnFull eObj ) {
    // käyttää pushOnFull-luokan value()-jäsenfunktiota
    // See Section 11.3
    cerr << "trying to push the value " << eObj.value()
        << " on a full stack\n";
}
catch ( Excp ) {
    // käyttää kantaluokan print()-jäsenfunktiota
    Excp::print( "an exception was encountered" );
}
```

Kuten mainittiin kohdassa 11.3, catch-lauseiden etsintäprosessi heitetylle poikkeukselle ei käyttäydy lainkaan kuin funktion ylikuormituksen ratkaisu. Funktion ylikuormituksen ratkaisun aikana parhaiten elinkelpoista funktiota valittaessa otetaan huomioon kaikki kutsupaikassa näkyvissä olevat funktioehdokkaat. Poikkeuksen käsittelyn aikana löydetty catch-lause ei välttämättä ole sellainen, joka parhaiten vastaa heitettyä poikkeusta. Catch-lause, joka valitaan, on *ensimmäiseksi täsmäävä*; tarkoittaa ensimmäistä kohdattua catch-lausetta, joka voi käsitellä poikkeuksen. Tästä syystä catch-lauseiden luettelossa ei kaikkein erikoistunein catch-lause saa esiintyä ensimmäisenä.

Catch-lauseen poikkeusesittely (tarkoittaa esittelyä suluissa catch-avainsanan jälkeen) käyttäytyy melko samalla tavalla kuin funktioparametrin esittely. Edellisessä esimerkissämme poikkeusesittely muistuttaa arvovälitysparametria. Olio eObj alustetaan poikkeusolion kopion arvolla samalla tavalla kuin funktion arvovälitysparametri alustetaan vastaavan argumentin arvon kopiolla. Kuten funktioparametrien yhteydessä, myös catch-lauseen poikkeusesittely voidaan muuttaa viittausesittelyksi. Catch-lause viittaa silloin suoraan poikkeusolioon, jonka throw-lauseke on luonut sen sijaan, että se loisi oman paikallisen kopion. Samasta syystä, että luokkatyypin parametrit tulisi esitellä viittauksina suurten luokkaolioiden tarpeettoman kopioinnin estämiseksi, on parempi, jos myös luokkatyyppisten poikkeusten poikkeusesittelyt ovat viittauksia. Catch-lauseen käyttäytymisessä on eroja riippuen siitä, onko poikkeusesittely olio

vai viittaus (katsomme näitä seuraavan kohdan läpikäynnin aikana).

Luvussa 11 esitellään uudelleenheittolauseke, jota catch-lause käyttää välittääkseen poikkeuksen toiselle catch-lauseelle ylöspäin funktiokutsuluettelossa. Uudelleenheittolausekkeen yleinen muoto on seuraava:

```
throw;
```

Millainen on sellaisen uudelleenheittolausekkeen käyttäytyminen, joka on sijoitettu kantaluokkatyypin catch-lauseeseen? Mikä esimerkiksi on uudelleen heitetyn poikkeuksen tyyppi, jos mathFunc() heittää poikkeustyypin divideByZero?

```
void calculate( int parm ) {
    try {
        mathFunc( parm ); // heittää divideByZero-poikkeuksen
    }
    catch ( mathExcp mExcp ) {
        // käsittelee osittain poikkeuksen
        // ja heittää uudelleen poikkeusolion
        throw;
    }
}
```

Onko uudelleen heitetyn poikkeuksen tyyppi sama, jonka mathFunc() heittää (eli tyyppiä divideByZero) vai onko se catch-lauseen poikkeusesittelyn tyyppi (tyyppiä mathExcp)?

Muista, että throw-lauseke heittää *alkuperäisen* poikkeusolion. Koska alkuperäinen poikkeustyyppi oli divideByZero, on uudelleen heitetyn poikkeuksen tyyppi divideByZero. Catch-lauseessa mExcp-olio alustetaan MathExcp-kantaluokan aliolion divideByZero-poikkeusolion kopiolla. Tätä kopiota käsitellään vain catch-lauseessa eikä se ole alkuperäinen poikkeusolio, joka on uudelleen heitetty.

Sanokaamme, että poikkeushierarkiamme luokkatyypeillä on tuhoajat. Esimerkiksi:

```
class pushOnFull {
public:
    pushOnFull( int i ) : _value( i ) { }
    int value() { return _value; }
    ~pushOnFull(); // juuri esitelty tuhoaja
private:
    int _value;
};
```

Milloin tätä tuhoajaa kutsutaan? Jotta voimme vastata tähän kysymykseen, pitää catch-lausetta tutkia vielä tarkemmin:

```
catch ( pushOnFull eObj ) {
    cerr << "trying to push the value " << eObj.value()
        << " on a full stack\n";
}
```

Koska poikkeusesittely esittelee eObj-olion paikalliseksi catch-lauseelle ja koska pushOnFull-luokalla on tuhoaja, eObj tuhotaan, kun catch-lause päättyy. Mutta milloin tuhoajaa kutsutaan luodulle poikkeusoliolle, kun poikkeus heitetään?

Voit ehkä tehdä muutaman arvauksen. Eräs mahdollisuus on, kun catch-lauseeseen saavutaan. Toinen mahdollisuus on, kun catch-lause päättyy. Jos kuitenkin poikkeus tuhotaan jommassakummassa näistä kohdista, se on voitu ehkä tuhota liian aikaisin. Huomaatko, miksi? Jos catch-lause heittää poikkeuksen uudelleen ja välittää poikkeusolion ylempänä funktiokutsuketjussa olevalle catch-lauseelle, ei poikkeusoliota voida tuhota ennen kuin viimeinen poikkeuksen käsittelevä catch-lause on saavutettu. Tästä syystä poikkeusoliota ei tuhota ennen kuin lopullinen catch-lause tälle poikkeukselle päättyy.

19.2.4 Poikkeusoliot ja virtuaalifunktiot

Jos heitetty poikkeusolio on johdettua luokkatyyppiä ja se käsitellään kantaluokkatyypin catch-lausessa, ei catch-lause yleensä voi käyttää johdetun luokkatyypin piirteitä. Esimerkiksi jäsenfunktiota value(), joka on esitelty poikkeusluokassa pushOnFull, ei voi käyttää catch-lauseessa, joka käsittelee Excp-tyyppisiä poikkeuksia:

```
catch ( const Excp &eObj ) {
    // virhe: Excp:llä ei ole value()-jäsenfunktiota
    cerr << "trying to push the value " << eObj.value()
        << " on a full stack\n";
}
```

Voimme suunnitella uudelleen poikkeusluokkahierarkiamme niin, että määrittelemme virtuaalifunktioita, joita voidaan käyttää Excp-kantaluokan catch-lauseessa erikoistuneimpien jäsenfunktioiden käynnistämiseksi johdetuissa luokkatyypeissä. Esimerkiksi:

```
// uudet luokkamäärittelyt ja virtuaalifunktiot
class Excp {
public:
    virtual void print() {
        cerr << "An exception has occurred"
            << endl;
    }
};

class stackExcp : public Excp { };

class pushOnFull : public stackExcp {
public:
    virtual void print() {
        cerr << "trying to push the value " << _value
            << " on a full stack\n";
    }
    // ...
};
```

print()-funktiota voidaan sitten käyttää catch-lauseessa seuraavasti:

```
int main( ) {
    try {
        // iStack::push() heittää poikkeuksen pushOnFull
    }   catch ( Excp eObj ) {
        eobj.print(); // kutsuu virtuaalifunktiota
                // hups, kantaluokan ilmentymä käynnistetty
    }
}
```

Vaikka heitetty poikkeus on tyyppiä pushOnFull ja vaikka print() on virtuaalifunktio, lause eObj.print() tulostaa seuraavan rivin:

```
An exception has occurred
```

Kutsuttu print()-funktio on Excp-kantaluokan jäsenfunktio eikä johdetun pushOnFull-luokan kumoava funktio. Miksi ei kutsuta johdetun luokan print()-funktiota?

Muista, että catch-lauseen poikkeusesittely käyttäytyy melko samalla tavalla kuin parametriesittely. Kun catch-lauseeseen saavutaan ja koska poikkeusesittely esittelee olion, eObj alustetaan poikkeusolion Excp-kantaluokan aliolion kopiolla. eObj on Excp-tyyppinen eikä pushOnFull-tyyppinen olio. Jotta johdetun luokkaolion virtuaalifunktioita voitaisiin kutsua, pitää poikkeusesittelyn esitellä osoitin tai viittaus. Esimerkiksi:

```
int main( ) {
    try {
        // iStack::push() heittää poikkeuksen pushOnFull
    }
    catch ( const Excp &eObj ) {
        eobj.print(); // kutsuu virtuaalifunktiota
                // pushOnFull::print()
    }
}
```

Tämän esimerkin catch-lauseen poikkeusesittely on myös Excp-kantaluokan tyyppinen, mutta koska eObj on viittaus ja koska eObj viittaa pushOnFull-tyyppiseen poikkeusolioon, eObj-oliota voidaan käyttää virtuaalifunktioiden käynnistämiseen, jotka on määritelty pushOnFull-luokkatyypissä. Kun catch-lause kutsuu virtuaalista print()-funktiota, kutsutaan johdetun pushOnFull-luokan print()-funktiota ja ohjelma tulostaa seuraavan rivin:

```
trying to push the value 879 on a full stack
```

Tässä on toinen hyvä syy esitellä catch-lauseen poikkeusesittely viittauksena varmistukseksi, että poikkeusolion tyyppiin liittyvät virtuaalifunktiot käynnistetään asianmukaisesti.

19.2.5 Pinon aukikelaus ja tuhoajakutsut

Kun poikkeus heitetään, käynnistyy sen catch-lauseen etsintä, joka voi käsitellä poikkeuksen. Se alkaa siitä funktiosta, joka poikkeuksen heitti, ja etenee ylöspäin sisäkkäistä funktiokutsuketjua, kunnes catch-lause poikkeukselle löytyy. Tätä catch-lauseen etsintäprosessia funktiokutsuketjua ylöspäin kutsutaan nimellä *pinon aukikelaus*. Pinon aukikelaus esiteltiin ensimmäisen kerran kohdassa 11.3.

Pinon aukikelauksen aikana, kun funktiot päättyvät catch-lauseen etsinnän seurauksena funktiokutsuketjussa, päättyvät myös funktioiden tekemät varsinaiset tehtävät ennen aikojaan. Tämä ei ehkä ole hyvä asia, jos funktio vaatii resursseja (jos se esimerkiksi avaa tiedoston tai varaa hieman muistia vapaavarastosta) eikä sitä koskaan vapauteta.

Ei ole olemassa ohjelmointitekniikkaa, joka mahdollistaisi ohjelmoijan kiertää tämä rajoitus. Joka kerta pinon aukikelauksen aikana, kun yhdistetty lause eli lohko päättyy catch-lausetta etsittäessä ja jos päättyvässä lohkossa on luokkatyyppinen olio, pinon aukikelausprosessi kutsuu automaattisesti tuon luokkatyypin tuhoajaa oliolle, ennen kuin yhdistetty lause tai funktio päättyy. (Paikalliset oliot on kuvattu kohdassa 8.1.)

Esimerkiksi seuraava luokka kapseloi vapaan muistin hankinnan int-taulukolle sen muodostajaan ja tuon muistin vapautuksen sen tuhoajaan:

```
class PTR {
public:
    PTR()  { ptr = new int[ chunk ]; }
    ~PTR() { delete[] ptr; }
private:
    int *ptr;
};
```

Tämän tyypin paikallinen olio luodaan manip()-funktiossa ennen kuin mathFunc()-funktiota kutsutaan:

```
void manip( int parm ) {
    PTR localPtr;
    // ...
    mathFunc( parm ); // heittää poikkeuksen divideByZero
    // ...
}
```

Jos mathFunc() heittää poikkeustyypin divideByZero, pinon aukikelaus etenee ylöspäin funktiokutsuketjua, jotta se löytäisi catch-lauseen heitetylle poikkeukselle. Funktio manip() tutkitaan, kun pinon aukikelaus etenee. Koska kutsua mathFunc()-funktioon ei ole sijoitettu try-lohkoon, ei manip()-funktiosta etsitä catch-lausetta tälle poikkeukselle. Pinon aukikelaus etenee ylöspäin funktiokutsuketjua funktioon, joka käynnisti manip()-funktion. Kuitenkin ennen kuin manip() päättyy tähän käsittelemättömään poikkeukseen, pinon aukikelaus tuhoaa kaikki manip()-funktiolle paikalliset luokkatyyppiset oliot, jotka on luotu ennen kuin mathFunc()-funktiota on kutsuttu. Paikallinen localPtr-olio tuhotaan ennen kuin pinon aukikelaus etenee ylöspäin funk-

tiokutsuketjua ja vapauttaa vapaavaraston muistin, johon localPtr viittaa, ja estää näin muistin hupenemisen.

Tästä syystä sanomme, että C++:n poikkeuksen käsittely kunnioittaa ohjelmointitekniikkaa, joka tunnetaan nimellä “resurssin hankinta on alustamista; resurssin vapautus on tuhoamista”. Jos resurssi on toteutettu luokkana, resurssien hankinnan toimenpiteet on kapseloitu luokan muodostajaan ja resurssien vapauttamisen toimenpiteet luokan tuhoajaan, kuten PTR-luokassamme, tuhotaan funktiolle paikallinen tuon luokkatyyppinen olio automaattisesti, jos funktio päättyy käsittelemättömään poikkeukseen. Kaikki toimenpiteet, joiden pitää tapahtua hankittujen resurssien vapauttamiseksi, eivät jää toteutumatta pinon aukikelauksen aikana, kun nuo toimenpiteet kapseloidaan luokan muodostajaan ja kutsutaan paikallisille olioille.

Saatat muistaa auto_ptr-piirteen, joka esiteltiin kohdassa 8.4 ja on määritelty C++-vakiokirjastoon. Tämä piirre käyttäytyy melko samalla tavalla kuin PTR-luokkamme. Se kapseloi vapaan muistin hankinnan muodostajiinsa ja vapauttaa tuon muistin sen tuhoajassa. auto_ptr-piirteen käyttäminen yksittäisen olion varaamiseen vapaavarastosta takaa, että muisti vapautetaan asianmukaisesti, kun yhdistetty lause eli funktio päättyy käsittelemättömään poikkeukseen pinon takaisinkelauksen aikana.

19.2.6 Poikkeusmääritykset

Poikkeusmäärityksiä käyttämällä funktioesittely voi määrittää poikkeukset, joita funktio voi heittää suoraan tai epäsuoraan. Poikkeusmääritys on tae siitä, että funktio ei heitä yhtään sellaista poikkeusta, jota ei ole lueteltu poikkeusmäärityksessä. Poikkeusmääritykset esiteltiin ensimmäisen kerran kohdassa 11.4. On muutama asia, jotka pitää mainita koskien poikkeusmäärityksiä ja luokkatyyppejä.

Ensiksi, poikkeusmäärityksiä voidaan tehdä luokan jäsenfunktioille aivan kuten jäsenettömille funktioille. Kuten jäsenettömillä funktioilla myös jäsenfunktioilla poikkeusmääritys tulee funktion parametriluettelon jälkeen. Esimerkiksi bad_alloc-luokka C++-vakiokirjastossa on määritelty niin, että sen jäsenfunktioilla on tyhjä poikkeusmääritys throw(). Tämä ilmaisee, että sen jäsenfunktiot eivät taatusti heitä yhtään poikkeusta:

```
class bad_alloc : public exception {
    // ...
public:
    bad_alloc() throw();
    bad_alloc( const bad_alloc & ) throw();
    bad_alloc & operator=( const bad_alloc & ) throw();
    virtual ~bad_alloc() throw()
    virtual const char* what() const throw();
};
```

Huomaa, että jos jäsenfunktio on esitelty const- tai volatile-tyyppiseksi jäsenfunktioksi kuten edellisen esimerkin what(), poikkeusesittely tulee const- tai volatile-määreen jälkeen funktioesittelyssä.

Funktion kaikissa esittelyissä pitää poikkeusmääritystyyppien olla samat. Jos jäsenfunktio on määritelty luokkamäärittelyn ulkopuolelle, pitää sen määrittelyssä olla sama poikkeusmääritys kuin jäsenfunktion esittelyssä luokkamäärittelyssä. Esimerkiksi:

```
#include <stdexcept>
// <stdexcept> defines class overflow_error

class transport {
    // ...
public:
    double cost( double, double ) throw ( overflow_error );
    // ....
};

// virhe: poikkeusmääritys eroaa
//       luokan jäsenluettelon esittelystä
double transport::cost( double rate, double distance ) { }
```

Kantaluokan virtuaalifunktiolla voi olla poikkeusmääritys, joka eroaa jäsenfunktion poikkeusmäärityksestä kumoten sen johdetussa luokassa. Kuitenkin johdetun luokan virtuaalifunktion poikkeusmäärityksen pitää olla joko yhtä rajoittava tai rajoittavampi kuin kantaluokan virtuaalifunktion poikkeusmääritys. Esimerkiksi:

```
class Base {
public:
    virtual double f1( double ) throw ();
    virtual int f2( int ) throw ( int );
    virtual string f3( ) throw ( int, string );
    // ...
};

class Derived : public Base {
public:
    // virhe: poikkeusmääritys ei ole niin rajoittava
    //       kuin base::f1():n
    double f1( double ) throw ( string );

    // ok: samanlainen poikkeusmääritys kuin base::f2():n
    int f2( int ) throw ( int );

    // ok: johdettu f3() on rajoittavampi
    string f3( ) throw ( int );
    // ...
};
```

Miksi johdetun luokan jäsenfunktion poikkeusmäärityksen on oltava yhtä rajoittava tai rajoittavampi kuin kantaluokan funktion? Tämä varmistaa, että kun johdettua virtuaalifunktiota kutsutaan osoittimella kantaluokan tyyppiin, on taattu, että kutsu ei riko kantaluokan jäsenfunktion poikkeusmäärityksiä. Esimerkiksi:

```
// takaa, että ei heitä poikkeuksia
void compute( Base *pb ) throw()
{
    try {
        pb->f3( ); // saattaa heittää int- tai string-tyyppisen poikkeuksen
    }
    // käsittelee poikkeukset Base::f3():sta
    catch ( const string & ) { }
    catch ( int ) { }
}
```

f3():n esittely Base-luokassa takaa, että funktio saattaa heittää vain string- tai int-tyyppisiä poikkeuksia. Funktio compute() on ohjelmoitu hyödyntämään tätä takuuta ja se määrittelee catch-lauseet käsitelläkseen vain nämä poikkeukset. Koska f3() on Derived-luokassa rajoittavampi kuin f3() Base-luokassa, ohjelmoidessamme Base-luokan rajapintaa ei olettamuksiamme vastaan koskaan rikota.

Lopuksi, luvussa 11 mainitsimme, että tyyppikonversiota ei sallita heitetyn poikkeuksen tyypin ja poikkeusmäärityksen tyypin välille. Tähän sääntöön on olemassa pieni poikkeus: kun poikkeusmääritys on luokkatyyppi tai osoitin luokkatyyppiin. Jos poikkeusmääritys määrittää luokan, silloin funktio saattaa heittää poikkeusmäärityksen mukaisen luokkatyypin, julkisesti johdetun luokan poikkeusolion. (Samalla tavalla osoitinten yhteydessä, jos poikkeusmääritys määrittää osoittimen luokkaan, funktio voi heittää poikkeusolioita, jotka ovat osoittimia tämän luokkatyypin julkisesti johdettuun luokkaan.) Esimerkiksi:

```
class stackExcp : public Excp { };
class popOnEmpty : public stackExcp { };
class pushOnFull : public stackExcp { };

void stackManip() throw( stackExcp )
{
    // ...
}
```

Poikkeusmääritys ei ainoastaan ilmaise, että stackManip() saattaa heittää stackExcp-tyyppisen poikkeuksen, vaan että se voi heittää myös popOnEmpty- tai pushOnFull-tyyppisen poikkeuksen. Muista, että kantaluokasta julkisesti johdettu luokka noudattaa *is-a*-suhdetta ja on erikoistunut ilmentymä sen yleisemmästä kantaluokasta. Koska popOnEmpty- ja pushOnFull-poikkeukset ovat eräänlaisia stackExcp-poikkeuksia, nämä poikkeukset eivät riko stackManip()-funktion poikkeusmäärityksiä vastaan.

19.2.7 Muodostajat ja funktion try-lohkot

On mahdollista esitellä funktio niin, että sen koko runko on try-lohkon sisällä. Sellaista try-lohkoa kutsutaan *funktion try-lohkoksi* (käsittelimme try-lohkoja ensimmäisen kerran kohdassa 11.2). Esimerkiksi:

```
int main()
try {
    // main():in funktiorunko
}
catch ( pushOnFull ) {
    // ...
}
catch ( popOnEmpty ) {
    // ...
}
```

Funktion try-lohkoon liittyy ryhmä catch-lauseita funktiorungossa. Jos lause heittää poikkeuksen funktiorungossa, funktiorungon jälkeen olevat käsittelijät otetaan huomioon poikkeuksen käsittelyä varten.

Funktion try-lohkosta tulee tarpeellinen luokan muodostajien yhteydessä. Tutkikaamme, miksi. Muodostajan määrittelyn muoto on seuraava:

```
class_name( parameter_list )
// jäsenen alustusluettelo:
:  member1( expression1 ) , // jäsenen 1 alustus
   member2( expression2 )   // jäsenen 2 alustus
// funktion runko:
{ /* ... */ }
```

`expression1` ja `expression2` voivat olla mitä tahansa lausekkeita. Erityisesti nämä lausekkeet voivat kutsua funktioita, jotka heittävät poikkeuksia.

Käyttäkäämme uudelleen luvussa 14 määriteltyä Account-luokkaa näyttääksemme todenmukaisemman esimerkin. Account-luokan muodostaja voidaan määritellä uudelleen seuraavasti:

```
inline Account::
Account( const char* name, double opening_bal )
    : _balance( opening_bal - ServiceCharge() )
{
    _name = new char[ strlen(name)+1 ];
    strcpy( _name, name );

    _acct_nmbr = get_unique_acct_nmbr();
}
```

Funktio `ServiceCharge()`, jota jäsenen alustuksessa kutsutaan `_balance`-jäsenelle, saattaa heittää poikkeuksen. Miten tämä muodostaja tulisi toteuttaa, jos haluamme käsitellä kaikki poikkeukset, jotka heitetään kutsutuista funktioista Account-tyyppisen olion muodostamisen aikana?

Try-lohkon sijoittaminen funktion rungon sisälle ei toimi. Esimerkiksi:

```
inline Account::
Account( const char* name, double opening_bal )
    : _balance( opening_bal - serviceCharge() )
{
    try {
        _name = new char[ strlen(name)+1 ];
        strcpy( _name, name );

        _acct_nmbr = get_unique_acct_nmbr();
    }
    catch ( ...) {
        // erikoiskäsittely
        // ei sieppaa poikkeuksia
        // jäsenen alustusluettelosta
    }
}
```

Koska try-lohko ei ympäröi jäsenen alustusluetteloa, ei muodostajan lopussa olevia catch-lauseita oteta huomioon, kun poikkeus heitetään jäsenen alustusluettelon serviceCharge()-funktiosta.

Funktion try-lohkon käyttö on ainoa ratkaisu, joka takaa, että kaikki olion muodostamisen aikana heitetyt poikkeukset siepataan muodostajassa. Funktion try-lohko voidaan määritellä Account-luokan muodostajalle seuraavasti:

```
inline Account::
Account( const char* name, double opening_bal )
try
    : _balance( opening_bal - serviceCharge() )
{
    _name = new char[ strlen(name)+1 ];
    strcpy( _name, name );
    _acct_nmbr = get_unique_acct_nmbr();
}
catch ( ... )
{
    // erikoiskäsittely
    // sieppaa nyt poikkeuksen ServiceCharge()-funktiosta
}
```

Huomaa, että avainsana try *edeltää* jäsenen alustusluetteloa ja try-lohkon yhdistetty lause ympäröi muodostajan funktion runkoa. Catch-lause catch(...) otetaan nyt huomioon poikkeusten käsittelyssä, jotka heitetään joko jäsenen alustusluettelosta tai muodostajan rungosta.

19.2.8 Poikkeusluokkahierarkia C++-vakiokirjastossa

Tämän kohdan alussa esittelimme poikkeusluokkahierarkian, jota käytettiin ohjelmiemme poikkeavuuksien raportoimiseen. C++-vakiokirjastossa on myös poikkeusluokkahierarkia. Poikkeusluokkia käytetään ohjelman poikkeavuuksien raportoimiseen, kun niitä kohdataan C++-vakiokirjaston funktioissa. Näitä poikkeusluokkia voidaan käyttää myös kirjoittamissamme ohjelmissa tai johtaa edelleen kuvaamaan poikkeuksia tarkemmin.

C++-vakiokirjaston poikkeusluokkahierarkian juuriluokan nimi on exception. Tämä luokka on määritelty vakio-otsikkotiedostoon <exception> ja on jokaisen C++-vakiokirjaston funktion heittämän poikkeuksen kantaluokka. Poikkeusluokan rajapinta on seuraava:

```
namespace std {
    class exception {
    public:
        exception() throw();
        exception( const exception & ) throw();
        exception& operator=( const exception& ) throw();
        virtual ~exception() throw();
        virtual const char* what() const throw();
    };
}
```

Kuten jokainen C++-vakiokirjastoon määritelty luokka, myös poikkeusluokka on sijoitettu std-nimiavaruuteen estämään globaalin nimialueen saastumista ohjelmissamme.

Neljä ensimmäistä funktiota luokkamäärittelyssä ovat oletusmuodostaja, kopiointimuodostaja, kopioinnin sijoitusoperaattori ja tuhoaja. Koska nämä jäsenfunktiot ovat julkisia, mikä tahansa ohjelma voi vapaasti luoda, kopioida ja sijoittaa poikkeusolioita. Tuhoaja on virtuaalifunktio, jotta exception-luokasta voitaisiin johtaa edelleen luokkamäärittelyjä.

Kiinnostavin funktio tässä luettelossa on what(), joka palauttaa C-tyylisen merkkijonon. Tämän C-tyylisen merkkijonon tarkoitus on antaa jonkinlainen tekstikuvaus heitetystä poikkeuksesta. what() on virtuaalifunktio. Luokat, jotka johdetaan exception-luokasta, voivat kumota what()-funktion omalla versiollaan, joka kuvaa paremmin johdettua poikkeusoliota.

Huomaa, että kaikilla poikkeusluokkamäärittelyn funktioilla on tyhjä poikkeusmääritys throw(). Tämä ilmaisee, että poikkeusluokan jäsenfunktiot eivät heitä yhtään poikkeusta. Ohjelma voi käsitellä poikkeusolioita (esimerkiksi poikkeustyypin mukaisessa catch-lauseessa) huolehtimatta siitä, että funktiot, jotka luovat, kopioivat ja tuhoavat poikkeusolioita, eivät heitä poikkeuksia.

Poikkeusten juuriluokan lisäksi C++-vakiokirjastossa on luokkia, joita voidaan käyttää ohjelmissamme poikkeavuuksien raportoimiseen. Virhemallissa, joka on vaikuttanut näihin esimääriteltyihin luokkiin, virheet on jaettu kahteen laveaan kategoriaan: *loogisiin virheisiin* ja *suorituksenaikaisiin virheisiin*.

Looginen virhe on sellainen, joka johtuu ohjelmamme sisäisestä logiikasta. Loogiset virheet ovat luultavasti estettävissä olevia virheitä, jotka voidaan havaita ennen kuin ohjelma aloittaa suorituksensa. Esimerkiksi rikkomukset loogisia alkutilaolettamuksia vastaan ovat loogisia virheitä. Loogiset virheet on määritelty C++-vakiokirjastoon seuraavasti:

```
namespace std {
    class logic_error : public exception {
    public:
        explicit logic_error( const string &what_arg );
    };
    class invalid_argument : public logic_error {
    public:
        explicit invalid_argument( const string &what_arg );
    };
    class out_of_range : public logic_error {
    public:
        explicit out_of_range( const string &what_arg );
    };
    class length_error : public logic_error {
    public:
        explicit length_error( const string &what_arg );
    };
    class domain_error : public logic_error {
    public:
        explicit domain_error( const string &what_arg );
    };
}
```

Funktio saattaa heittää invalid_argument-poikkeuksen, jos se saa kelpaamattoman argumenttiarvon, kun taas funktio saattaa heittää out_of_range-poikkeuksen, jos se saa argumentin, jonka arvo ei ole odotetulla arvoalueella. Funktio saattaa heittää length_error-poikkeuksen raportoidakseen yrityksestä tuottaa olio, jonka pituus ylittää suurimman sallitun koon. Toteutus voi heittää domain_error-poikkeuksen raportoidakseen sovellustasoisista virheistä.

Vastakohtana suorituksenaikainen virhe johtuu tapahtumasta, joka menee ohjelman viittausalueen ulkopuolelle. Suorituksenaikainen virhe on luultavasti havaittavissa vain ohjelman ollessa suorituksessa. Suorituksenaikaiset virheet on määritelty C++-vakiokirjastoon seuraavasti:

```
namespace std {
    class runtime_error : public exception {
    public:
        explicit runtime_error( const string &what_arg );
    };
    class range_error : public runtime_error {
    public:
        explicit range_error( const string &what_arg );
    };
    class overflow_error : public runtime_error {
```

```
    public:
        explicit overflow_error( const string &what_arg );
    };
    class underflow_error : public runtime_error {
    public:
        explicit underflow_error( const string &what_arg );
    };
}
```

Funktio saattaa heittää range_error-poikkeuksen raportoidakseen raja-arvovirheistä sisäisissä laskennoissa. Funktio saattaa heittää overflow_error-poikkeuksen raportoidakseen aritmeettisesta ylivuotovirheestä ja underflow_error-poikkeuksen aritmeettisesta alivuotovirheestä.

Poikkeusluokka on myös new()-operaattorin heittämän bad_alloc-poikkeuksen kantaluokka, kun operaattori epäonnistuu varaamaan pyydettyä muistitilaa (kuten mainitsimme kohdassa 8.4) sekä myös heitetyn bad_cast-poikkeuksen kantaluokka, kun viittauksen dynaaminen tyyppimuunnos (dynamic_cast) epäonnistuu (kuten mainitsimme kohdassa19.1).

Määritelkäämme uudelleen operator[]()-operaattori, joka määriteltiin Array-luokkamallille kohdassa 16.12, niin että se heittää range_error-tyyppisen poikkeuksen, jos Array:n indeksi on rajojensa ulkopuolella:

```
#include <stdexcept>
#include <string>

template <class elemType>
class Array {
public:
    // ...

    elemType& operator[]( int ix ) const
    {
        if ( ix < 0 || ix >= _size )
        {
            string eObj =
            "out_of_range error in Array<elemType>::operator[]()";

            throw out_of_range( eObj );
        }
        return _ia[ix];
    }

    // ...
private:
    int _size;
    elemType *_ia;
};
```

Jotta esimääriteltyjä poikkeusluokkia voisi käyttää, pitää ottaa mukaan otsikkotiedosto <stdexcept>. String-tyyppinen eObj-olio, joka välitetään out_of_range-muodostajalle, kuvaa heitettyä poikkeusta. Tämä tieto voidaan hakea, kun poikkeus on siepattu ja käytetään poikkeuksen what()-jäsenfunktiota kuten seuraavassa:

```
int main()
{
    try {
        // main()-funktio kuten määriteltiin kohdassa 16.2
    }
    catch ( const out_of_range &excp ) {
        // tulostaa:
        // out_of_range error in Array<elemType>::operator[]()
        cerr << excp.what() << "\n";
        return -1;
    }
}
```

Tässä toteutuksessa yli rajojensa menevä indeksi try_array()-funktiossa saa Array-operator[]()-operaattorin heittämään out_of_range-tyyppisen poikkeuksen main()-funktioon.

Harjoitus 19.5

Mitä poikkeuksia seuraavat funktiot saattavat heittää?

```
#include <stdexcept>

(a) void operate() throw( logic_error );
(b) int mathOper( int ) throw( underflow_error, overflow_error );
(c) char manip( string ) throw( );
```

Harjoitus 19.6

Kerro, kuinka C++:n poikkeuksen käsittely tukee ohjelmointitekniikkaa, joka tunnetaan nimellä “resurssin hankinta on alustamista; resurssin vapautus on tuhoamista.”

Harjoitus 19.7

Miksi catch-lauseiden luettelo try-lohkon jälkeen on virheellinen? Kuinka korjaisit sen?

```
#include <stdexcept>

int main() {
    try {
        // C++-vakiokirjaston käyttö
    }
    catch( exception ) {
    }
    catch( const runtime_error &re ) {
    }
```

```
    catch( overflow_error eobj ) {
    }
}
```

Harjoitus 19.8

Olkoon seuraava C++-perusohjelma:

```
int main() {
    // C++-vakiokirjaston käyttö
}
```

Muokkaa main()-funktiota niin, että se sieppaa kaikki C++-vakiokirjaston funktioiden heittämät poikkeukset. Käsittelijöiden tulisi tulostaa virheeseen liittyvä virheilmoitus ennen abort()-funktion kutsua (määritelty otsikkotiedostossa <cstdlib>) main()-funktion lopettamiseksi.

19.3 Ylikuormituksen ratkaisu ja periytyminen ◆

Luokan periytyminen vaikuttaa kaikkiin funktion ylikuormituksen ratkaisun vaiheisiin. Muista, että funktion ylikuormituksen ratkaisun kolme vaihetta ovat seuraavat:

1. Valitse funktioehdokkaat.
2. Valitse elinkelpoiset funktiot.
3. Valitse parhaiten elinkelpoinen funktio.

(Katso kohdasta 9.2 koko käsittely.)

Periytyminen vaikuttaa funktioehdokkaitten valintaan, koska kantaluokkiin liittyvät funktiot, joko niiden jäsenfunktiot tai funktiot, jotka on määritelty nimiavaruuksiin, joissa kantaluokat on määritelty, otetaan huomioon funktioehdokkaita valittaessa. Periytyminen vaikuttaa elinkelpoisten funktioiden valintaan, koska suurempi joukko käyttäjän määrittelemiä konversioita otetaan huomioon, kun huomioidaan argumenttien konversioita elinkelpoisten funktioiden parametrien tyypeille. Periytyminen vaikuttaa parhaiten elinkelpoisen funktion valintaan, koska se vaikuttaa konversiosarjan paremmuusjärjestykseen, joita voidaan käyttää argumentille sen konvertoimiseksi funktion parametrin tyyppiseksi. Tässä kohdassa tutkimme periytymisen vaikutusta kolmeen funktion ylikuormituksen ratkaisun vaiheeseen tarkemmin.

19.3.1 Funktioehdokkaat

Periytyminen vaikuttaa funktion ylikuormituksen ratkaisun ensimmäiseen vaiheeseen — siis kutsun funktioehdokkaitten valitsemiseen. Periytymisen vaikutus ensimmäiseen vaiheeseen vaihtelee riippuen siitä, onko se tavallinen funktion kutsu muotoa

```
func( args );
```

vai onko se jäsenfunktion kutsu, jossa käytetään jäsenen käsittelyoperaattoria, pistettä tai nuolta:

```
object.memfunc( args );
pointer->memfunc( args );
```

Tässä alikohdassa katsomme, kuinka periytyminen vaikuttaa kumpaankin tilanteeseen vuorollaan.

Kun tavallisessa funktion kutsussa argumentti on luokkatyyppi, viittaus luokkatyyppiin tai osoitin luokkatyyppiin, ja luokkatyyppi on määritelty nimiavaruuteen, funktiot, jotka on esitelty tuossa nimiavaruudessa samalla nimellä kuin kutsuttu funktio, ovat funktioehdokkaita, vaikka nuo funktiot eivät ole näkyvissä kutsupaikassa (tämä esiteltiin tarkemmin kohdassa 15.10). Jos periytymisessä argumentti on luokkatyyppinen, viittaus luokkatyyppiin tai osoitin luokkatyyppiin ja luokalla on kantaluokkia, funktiot, jotka on esitelty nimiavaruuteen, jossa kantaluokat on määritelty samalla nimellä kuin kutsuttu funktio, lisätään myös ne funktioehdokkaitten joukkoon. Esimerkiksi:

```
namespace NS {
    class ZooAnimal { /* ... */ };
    void display( const ZooAnimal& );
}

// Bear:in kantaluokka on esitelty nimiavaruudessa NS
class Bear : public NS::ZooAnimal { };

int main() {
    Bear baloo;

    display( baloo );
    return 0;
}
```

`baloo`-argumentin luokkatyyppi on Bear. `display()`-kutsun funktioehdokkaat eivät ole vain ne funktiot esittelyineen, jotka ovat näkyvissä `display()`-funktion kutsupaikassa, vaan myös funktiot nimiavaruudessa, jossa Bear-luokka ja sen ZooAnimal-kantaluokka on esitelty. Funktio `display(const ZooAnimal&)`, joka on esitelty NS-nimiavaruudessa, lisätään funktioehdokkaitten joukkoon.

Jos argumentti on luokkatyyppi ja luokkamäärittelyssä esitellään ystäväfunktioita samalla nimellä kuin funktio, jota kutsuttiin, ovat nämä ystäväfunktiot funktioehdokkaita, vaikka näiden ystäväfunktioiden esittelyt eivät näkyisikään kutsupaikkaan (kuten esitettiin kohdassa 15.10). Jos periytymisessä argumentti on luokkatyyppi, jolla on kantaluokkia, ystäväfunktiot, joilla on sama nimi kuin funktiolla, jota kutsuttiin ja on esitelty kantaluokan määrittelyssä, lisätään myös funktioehdokkaitten joukkoon. Esitelkäämme esimerkiksi aikaisemmin näytetty `display()`-funktio ystäväfunktioksi ZooAnimal-luokkaan:

```
namespace NS {
    class ZooAnimal {
        friend void display( const ZooAnimal& );
    };
```

```
}

// Bear:in kantaluokka on esitelty nimiavaruuteen NS
class Bear : public NS::ZooAnimal { };

int main() {
    Bear baloo;

    display( baloo );
    return 0;
}
```

Funktioargumentin baloo tyyppi on Bear. Sen ZooAnimal-kantaluokassa esitellään display()-funktio ystävänä. display() on NS-nimiavaruuden jäsen, vaikka sitä ei koskaan ole esitelty siellä suoraan. Normaali etsintä NS-nimiavaruudesta ei löydä tätä ystäväfunktiota. Koska kuitenkin display()-funktion kutsussa on argumenttina Bear-tyyppi, lisätään Bear:in ZooAnimal-kantaluokassa esitelty funktio funktioehdokkaitten joukkoon.

Siten, jos tavallisen funktion kutsussa on argumentti, joka on joko luokkatyyppinen olio, osoitin luokkatyyppiin tai viittaus luokkatyyppiin, funktioehdokkaat ovat yhdiste seuraavista:

1. Funktiot, jotka ovat näkyvissä kutsupaikassa.
2. Funktiot, jotka on esitelty nimiavaruudessa, jossa luokkatyyppi on esitelty, tai nimiavaruuksissa, joissa luokan kantaluokat on määritelty.
3. Funktiot, jotka ovat luokan ystäviä tai luokan kantaluokkien ystäviä.

Periytyminen vaikuttaa myös funktioehdokkaitten joukkoon, joka on muodostettu jäsenfunktioista, joissa käytetään piste- tai nuolioperaattoria jäsenen käsittelyssä. Kuten näimme kohdassa 18.4, jäsenfunktion esittely johdetussa luokassa ei ylikuormita samannimisiä jäsenfunktioita, jotka on esitelty kantaluokissa. Sen sijaan johdetun luokan jäsenfunktio piilottaa samannimiset jäsenfunktiot kantaluokista, vaikka funktioiden parametriluettelot olisivat erilaisia. Esimerkiksi:

```
class ZooAnimal {
public:
    Time feeding_time( string );
    // ...
};
class Bear : public ZooAnimal {
public:
    // peittää ZooAnimal::feeding_time( string )
    Time feeding_time( int );
    // ....
};

Bear Winnie;

// virhe: ZooAnimal::feeding_time( string ) peittyy
```

```
Winnie.feeding_time( "Winnie" );
```

Jäsenfunktio feeding_time(int), joka on esitelty Bear-luokassa, piilottaa jäsenfunktion feeding_time(string), joka on esitelty Bear:in ZooAnimal-kantaluokassa. Koska jäsenfunktion kutsu tapahtuu Bear-tyyppisen Winnie-olion kautta, vain Bear-luokan viittausalueelta etsitään funktioehdokkaita jäsenfunktiokutsua varten. Ainoa Bear-luokan viittausalueella näkyvä esittely on feeding_time(int). Täten feeding_time(int) on ainoa funktioehdokas jäsenfunktiokutsua varten, joten kutsu on virheellinen.

Jotta tämä tilanne voitaisiin korjata ja saada kantaluokan jäsenfunktiot kuormittamaan johdetun luokan jäsenfunktioita, johdetun luokan suunnittelija voi esitellä kantaluokan jäsenfunktiot johdetun luokan viittausalueelle using-esittelyillä. Esimerkiksi:

```
class Bear : public ZooAnimal {
public:
    // feeding_time( int ) ylikuormitetaan ZooAnimal-kantaluokasta
    using ZooAnimal::feeding_time;
    Time feeding_time( int );
    // ...
};
```

Nyt kaksi feeding_time()-funktiota ovat johdetun Bear-luokan viittausalueella. Molemmat funktiot ovat nyt osa kutsun funktioehdokkaita

```
// ok: ZooAnimal::feeding_time( string ) tulee kutsutuksi
Winnie.feeding_time( "Winnie" );
```

ja kantaluokan funktio feeding_time(string) valitaan kutsua varten.

Kun moniperiytymisessä kootaan jäsenfunktioehdokkaita, pitää jäsenfunktioiden esittelyiden löytyä samasta kantaluokasta, tai kutsu on virheellinen. Esimerkiksi:

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

class Bear : public ZooAnimal {
public:
    void print( );
    using ZooAnimal::feeding_time;
    Time feeding_time( int );
    // ...
};

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

```
int main()
{
    Panda yin_yang;

    // virhe: moniselitteinen: jompikumpi näistä:
    //      Bear::print()
    //      Endangered::print( ostream& )
    yin_yang.print( cout );

    // ok: kutsuu Bear::feeding_time()
    yin_yang.feeding_time( 56 );
}
```

Kun etsitään esittelyä jäsenfunktiolle print() Panda-luokan viittausalueelta, sekä Bear::print() että Endangered::print() löytyvät. Koska print()-funktiolle löytyneet esittelyt eivät ole samassa kantaluokassa ja vaikka molemmilla funktioilla on erilaiset parametriluettelot, jää kutsun funktioehdokasjoukko tyhjäksi ja jäsenfunktiokutsu on virheellinen. Tämän virheen korjaamiseksi pitää Panda-luokan esitellä sen oma print()-funktio. Kun etsitään esittelyä jäsenfunktiolle feeding_time() Panda-luokan viittausalueelta, sekä ZooAnimal::feeding_time() että Bear::feeding_time() löytyvät Bear-luokan viittausalueelta. Koska molemmat esittelyt löytyvät samasta kantaluokasta, funktioehdokasjoukko sisältää molemmat funktiot ja jäsenfunktio Bear::feeding_time() valitaan kutsua varten.

19.3.2 Elinkelpoiset funktiot ja käyttäjän määrittelemät konversiosarjat

Periytyminen vaikuttaa funktion ylikuormituksen ratkaisun myös toiseen vaiheeseen, jossa valitaan funktioehdokkaitten joukosta elinkelpoisia funktioita, joita voidaan kutsua. Elinkelpoinen funktio on sellainen, jossa jokaiselle funktiokutsun argumentille löytyy tyyppikonversio, jolla argumentit saadaan elinkelpoisen funktion vastaavien parametrien tyyppisiksi.

Kohdassa 15.9 kuvataan, kuinka luokan suunnittelija voi tehdä käyttäjän määrittelemiä konversioita luokkatyyppisille olioille. Kääntäjä käynnistää automaattisesti nämä käyttäjän määrittelemät konversiot, jotka konvertoivat funktiokutsun argumentin elinkelpoisen funktion vastaavaksi parametriksi. Käyttäjän määrittelemä konversio on joko konversiofunktio tai eieksplisiittiinen muodostaja, joka saa yhden argumentin. Periytymisessä otetaan huomioon suurempi joukko käyttäjän määrittelemiä konversiota funktion ylikuormituksen ratkaisuun.

Konversiofunktiot peritään aivan kuten muutkin luokan jäsenfunktiot. Saatamme esimerkiksi päättää määritellä konversiofunktion ZooAnimal-luokalle seuraavasti:

```
class ZooAnimal {
public:

    // konversio: ZooAnimal ==> const char*
    operator const char*();
```

```
    // ...
};
```

Johdettu Bear-luokka perii tämän konversiofunktion ZooAnimal-kantaluokaltaan. Aina, kun Bear-tyyppistä arvoa käytetään siellä, missä edellytetään tyyppiä const char*, konversiofunktiota kutsutaan automaattisesti konvertoimaan Bear-arvo const char* -tyyppiseksi. Esimerkiksi:

```
extern void display( const char* );

Bear yogi;

// ok: yogi ==> const char*
display( yogi );
```

Ei-eksplisiittiset muodostajat, jotka saavat yhden argumentin, muodostavat toisen automaattisen konversiojoukon. Muodostaja voi konvertoida parametrinsa arvon luokkansa tyypiksi. Saatamme esimerkiksi päättää määritellä muodostajan ZooAnimal-luokalle seuraavasti:

```
class ZooAnimal {
public:
    // konversio: int ==> ZooAnimal
    ZooAnimal( int );

    // ...
};
```

Tätä muodostajaa voidaan käyttää konvertoitaessa kokonaislukuarvo ZooAnimal-tyyppiseksi arvoksi. Muodostajia ei kuitenkaan peritä. ZooAnimal-muodostajaa ei voida käyttää olion konvertoimiseen, kun tarvitaan ZooAnimal-luokasta johdettua luokkatyyppiä. Esimerkiksi:

```
const int cageNumber = 8788;

void mumble( const Bear & );

// virhe: tätä ei käytetä: ZooAnimal( int )
mumble( cageNumber );
```

Koska konversion kohteen luokkatyyppi on Bear, joka on mumble()-funktion parametrin tyyppi, otetaan vain Bear-luokan muodostajat huomioon.

19.3.3 Parhaiten elinkelpoinen funktio

Periytyminen vaikuttaa myös funktion ylikuormituksen ratkaisun kolmanteen vaiheeseen, jossa valitaan parhaiten elinkelpoinen funktio. Jotta parhaiten elinkelpoinen funktio voidaan valita, laitetaan ne tyyppikonversiot paremmuusjärjestykseen, joilla argumentit voidaan konvertoida vastaaviksi funktion parametreiksi. Millainen on seuraavien automaattisten konversioiden paremmuusjärjestys?

1. Kun argumentti konvertoidaan johdetun luokan tyypistä minkä tahansa sen kantaluokan tyyppiseksi parametriksi.
2. Kun osoitin johdettuun luokkaan konvertoidaan osoittimeksi mihin tahansa sen kantaluokan tyyppiin.
3. Kun viittaus kantaluokan tyyppiin alustetaan johdetun luokkatyypin *lvaluella*.

Kun funktion argumentteihin käytettyjä konversioita laitetaan paremmuusjärjestykseen, on näillä konversioilla vakiokonversion paremmuusjärjestys. (Muut vakiokonversiot on kuvattu kohdassa 9.3.) Nämä konversiot eivät ole käyttäjän määrittelemiä konversioita, koska ne eivät riipu konversiofunktioista eivätkä luokan suunnittelijan määrittelemistä muodostajista. Esimerkiksi:

```
extern void release( const ZooAnimal& );
Panda yinYang;

// vakiokonversio: Panda -> ZooAnimal
release( yinYang );
```

Koska Panda-tyyppinen yinYang-argumentti alustaa viittauksen kantaluokan tyypillä, on konversiolla vakiokonversion paremmuusjärjestys.

Näimme kohdassa 15.10, että vakiokonversiosarja on parempi kuin käyttäjän määrittelemä konversiosarja, kun tyyppikonversioita laitetaan paremmuusjärjestykseen parhaiten elinkelpoista funktiota valittaessa. Esimerkiksi:

```
class Panda : public Bear,
              public Endangered
{
    // perii: ZooAnimal::operator const char *()
};

Panda yinYang;

extern void release( const ZooAnimal& );
extern void release( const char * );

// vakiokonversio: Panda -> ZooAnimal
// valitsee: release( const ZooAnimal& )
release( yinYang );
```

Sekä release(const char*) että release(const ZooAnimal&) ovat elinkelpoisia funktioita. Funktio release(const ZooAnimal&) on elinkelpoinen, koska argumentti voi alustaa viittausparametrin vakiokonversion kautta. Funktio release(const char*) on elinkelpoinen funktio, koska käyttäjän määrittelemä konversio, joka käyttää konversiofunktiota ZooAnimal::operator const char*(), voi konvertoida argumentin const char* -tyypiksi. Koska vakiokonversiosarja on parempi kuin käyttäjän määrittelemä konversiosarja, valitaan release(const ZooAnimal&)-funktio parhaiten elinkelpoiseksi funktioksi.

Kun eri vakiokonversioita johdetusta luokkatyypistä eri kantaluokkien tyypeiksi laitetaan

paremmuusjärjestykseen, konversiota kantaluokaksi, joka on lähempänä johdettua luokkatyyppiä, pidetään parempana vakiokonversiona kuin konversiota kantaluokaksi, joka on kauempana johdetusta luokkatyypistä. Esimerkiksi seuraava kutsu ei ole moniselitteinen, vaikka vakiokonversiota vaaditaan molemmissa tapauksissa. Konversiota Bear-kantaluokaksi pidetään parempana kuin konversiota ZooAnimal-kantaluokaksi, koska Bear-luokka on lähempänä johdettua Panda-luokkaa. Parhaiten elinkelpoinen funktio kutsua varten on siis release(const Bear&):

```
extern void release( const ZooAnimal& );
extern void release( const Bear& );

// ok: release( const Bear& );
release( yinYang );
```

Sama sääntö pätee myös osoittimiin. Kun laitetaan paremmuusjärjestykseen eri vakiokonversioita, jotka tehdään johdetun luokan tyyppisestä osoittimesta eri kantaluokkien tyyppisiksi osoittimiksi, pidetään parempana sitä konversiota, joka on lähimpänä johdetun luokan tyyppiä. Samaa sääntöä laajennetaan myös void*-tyypin käsittelyyn. Vakiokonversio osoittimelle kantaluokkaan on parempi kuin konversio void*-tyypiksi. Olkoon esimerkkinä seuraava pari ylikuormitettuja funktioita:

```
void receive( void* );
void receive( ZooAnimal* );
```

Funktio receive(ZooAnimal*) on parhaiten elinkelpoinen funktio argumenttityypille Panda*.

Moniperiytyminen voi saada aikaan, että kaksi vakiokonversiota johdetusta luokkatyypistä eri kantaluokkien tyypeiksi ovat yhtä hyviä, jos molemmat kantaluokat ovat yhtä etäällä johdetusta luokkatyypistä. Esimerkiksi Panda on johdettu sekä Bea- että Endangered-luokasta. Nämä kantaluokat ovat yhtä etäällä johdetusta Panda-luokasta, jolloin konversiot Panda-luokan oliosta molempiin näihin kantaluokkiin ovat yhtä hyviä. Koska molemmat konversiot ovat yhtä hyviä, ei parhaiten elinkelpoista funktiota voida valita seuraavalle kutsulle, jolloin se on virheellinen:

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

/* virhe: moniselitteinen kutsu:
 * mumble():n valinnat:
 *    void mumble( const Bear & );
 *    void mumble( const Endangered & );
 */
mumble( yinYang );
```

Jotta kutsu voitaisiin ratkaista, pitää ohjelmoijan käyttää pakotettua tyyppimuunnosta:

```
mumble( static_cast< Bear >( yinYang ) ); // ok
```

Johdetun luokan olion alustamiseen kantaluokkatyyppisellä oliolla, johdetun luokan tyyppisen viittauksen alustamiseen kantaluokkatyyppisellä oliolla tai konversioon osoittimesta kan-

taluokkatyyppiin osoittimeksi johdetun luokan tyyppiin ei koskaan käytetä automaattista konversiota. (Sellaiset konversiot voidaan kuitenkin tehdä käyttämällä pakotettua, dynaamista tyyppimuunnosta (dynamic_cast) kuten näimme kohdassa 19.1.) Seuraavalle kutsulle ei löydy elinkelpoisia funktioita, koska automaattista konversiota ei ole olemassa ZooAnimal-argumenttityypin konvertoimiseksi johdetun luokan tyyppiseksi:

```
extern void release( const Bear& );
extern void release( const Panda& );

ZooAnimal za;

// virhe: ei täsmää
release( za );
```

Seuraavassa esimerkissä kutsulle parhaiten elinkelpoinen funktio on release(const char*). Tämä voi näyttää yllättävältä, koska argumentin konvertoimiseen funktion parametrin tyyppiseksi käytetty konversio on käyttäjän määrittelemä konversiosarja, joka käyttää ZooAnimal:in konversiofunktiota char*(). Koska kuitenkaan ei ole olemassa automaattista konversiota kantaluokan tyypistä johdetun luokan tyypiksi, ei release(const Bear&) ole elinkelpoinen funktio ja release(const char*) on ainoa elinkelpoinen funktio kutsulle:

```
class ZooAnimal {
public:
    // konversio: ZooAnimal ==> const char*
    operator const char*();

    // ...
};

extern void release( const char* );
extern void release( const Bear& );

ZooAnimal za;

// za ==> const char*
// ok: release( const char* )
release( za );
```

Harjoitus 19.9

Olkoon seuraava luokkahierarkia, jossa on seuraavat jäsenfunktiot:

```
class Base1 {
public:
    ostream& print();
    void debug();
    void writeOn();
    void log( string );
    void reset( void *);
```

```
    // ...
};

class Base2 {
public:
    void debug();
    void readOn();
    void log( double );
    // ...
};

class MI : public Base1, public Base2 {
public:
    ostream& print();
    using Base1::reset;
    void reset( char * );
    using Base2::log;
    using Base1::log;
    // ...
};
```

Mitkä funktioista ovat jäsenfunktioehdokkaita seuraaville kutsuille?

```
MI *pi = new MI;
(a) pi->print();  (c) pi->readOn();  (e) pi->log( num );
(b) pi->debug();  (d) pi->reset(0);  (f) pi->writeOn();
```

Harjoitus 19.10

Olkoon seuraava luokkahierarkia, jossa on seuraavat konversiofunktiot:

```
class Base {
public:
    operator int();
    operator const char *();
    // ...
};
class Derived : public Base {
public:
    operator double();
    // ...
};
```

Mikä funktio, jos yksikään, valitaan parhaiten elinkelpoiseksi funktioksi seuraaville kutsuille? Luettele funktioehdokkaat, elinkelpoiset funktiot ja argumentin tyyppikonversiot jokaiselle elinkelpoiselle funktiolle.

```
(a) void operate( double );
    void operate( string );
    void operate( const Base & );

    Derived *pd = new Derived;
    operate( *pd );

(b) void calc( int );
    void calc( double );
    void calc( const Derived & );

    Base *pb = new Derived;
    operate( *pb );
```