Osa IV

Oliokeskeinen ohjelmointi

Osa 4 keskittyy oliokeskeiseen ohjelmointiin — tämä tarkoittaa C++:n luokkapiirteiden määrittelyä ja käyttöä, joilla voidaan luoda uusia tyyppejä ja joita voidaan käsitellä yhtä helposti kuin sisäisiä tyyppejä. Koska C++:ssa voidaan luoda uusia tyyppejä ongelma-alueiden kuvaamiseen, ohjelmoijien on mahdollista kirjoittaa helpommin ymmärrettäviä sovelluksia. Luokkapiirteet mahdollistavat ohjelmoijan pitää erillään uuden tyypin taustalla olevat yksityiskohdat rajapinnan määrittelystä ja operaatioista, jotka on tarkoitettu uuden tyypin käyttäjille. Tämän erottelun avulla tarvitsee pitää vähemmän huolta monista huomioitavista asioista, jotka saavat ohjelmoinnin vaivalloiseksi. Sovelluksen perustyypit voidaan toteuttaa kerran ja käyttää sitten uudestaan. Tiedon ja toimintojen kapseloinnin piirteet, jotka tukevat uuden tyypin toteutusta, voivat yksinkertaistaa sovellusten myöhempää ylläpitoa ja kehitystä dramaattisesti.

Luku 13 keskittyy yleiseen luokkamekanismiin: luokan määritelyyn, käsitteeseen *tiedon piilotus* (tarkoittaa luokan julkista rajapintaa ja yksityistä toteutusta) ja luokan olioilmentymien määrittelyyn. Lisäksi käsitellään sekä myös luokan viittausaluetta, sisäkkäisiä luokkia ja luokkia nimiavaruuden jäseninä. Luvussa 14 kerrotaan yksityiskohtaisesti erityistuesta, jota C++ tarjoaa luokkaolioiden alustamiseen, tuhoamiseen ja sijoittamiseen käyttämällä erityisiä jäsenfunktioita nimeltään *muodostaja, tuhoaja* ja *kopioinnin sijoitusoperaattori*. Katsomme myös jäsenten alustamista ja kopiointia, jossa luokkaolio alustetaan tai siihen sijoitetaan toisen samanlaisen luokkatyypin olion arvo.

Luvussa 15 katsotaan luokkakohtaista operaattorin ylikuormitusta. Operaattorin ylikuormitus mahdollistaa luokkatyyppisten operandien käytön luvussa 4 kuvattujen sisäisten operaattorien kanssa. Operaattorin ylikuormitus mahdollistaa luokkatyyppisten olioiden käytön yhtä helposti kuin sisäisten tyyppien käytön. Aluksi luvussa 15 esitellään operaattorin ylikuormituksen yleiset käsitteet, sitten katsotaan tiettyjä operaattoreita kuten sijoitus-, indeksi-, kutsu- ja luokkakohtaisia new- ja delete-operaattoreita. Joskus on tarpeellista esitellä ylikuormitettu operaattori ystävänä luokalle tietyin käyttöehdoin. Tämä luku kertoo, miksi ystävät ovat joskus tarpeellisia. Sitten luvussa esitellään toinen erityinen luokan jäsenfunktiokategoria, *konversiofunktiot*, joka mahdollistaa ohjelmoijan määritellä vakiokonversioita luokkatyypeille. Kääntäjä

käyttää näitä konversiofunktioita implisiittisesti, kun luokkaolioita on käytetty funktion argumentteina tai operandeina sisäisille tai ylikuormitetuille operaattoreille. Luku päättyy esitykseen funktion ylikuormituksen ratkaisun säännöistä, jotka koskevat luokan argumentteja, luokan jäsenfunktioita ja ylikuormitettuja operaattoreita.

Luokkamallit ovat luvun 16 aiheena. Luokkamalli on esikuvaus luokan luomiselle, jossa yksi tai useampi tyyppi tai arvo on parametroitu. Esimerkiksi vektoriluokka voi parametroida sisältämiensä elementtien tyypin. Puskuriluokka voi parametroida, ei vain sisältämiensä elementtien tyyppiä, vaan yhtä hyvin puskurinsa koon. Tässä luvussa kuvataan, kuinka luokkamalli määritellään ja kuinka luodaan erityisilmentymiä luokkamallista. C++-luokkien tukea katsellaan uudelleen mallien kannalta, jolloin jäsenfunktiot, ystävien esittelyt ja sisäkkäiset tyypit käsitellään. Luvussa palataan myös mallin käännösmuotoon, jota käsiteltiin luvussa 10, jotta nähtäisiin, kuinka se vaikuttaa luokkamalleihin.

13

Luokat

C++:n luokkamekanismi mahdollistaa sen, että käyttäjät voivat määritellä omia tietotyyppejä. Tästä syystä luokkia kutsutaan usein *käyttäjän määrittelemiksi tyypeiksi*. Luokka voi lisätä toiminnallisuutta jo olemassa olevaan tyyppiin — kuten luvussa 2 esiteltyyn IntArray-luokkaan, joka sisälsi enemmän toimintoja kuin int-taulukkotyyppi. Luokkia voidaan myös käyttää, kun halutaan esitellä aivan uusia tyyppejä kuten Screen- tai Account-luokka. Luokkia käytetään tyypillisesti sellaisten abstraktioiden määrittelyyn, joita ei voida luonnostaan yhdistää sisäisiin tietotyyppeihin.

Tässä luvussa näemme, kuinka luokkatyyppejä määritellään ja kuinka luokkaolioita käytetään. Näemme, kuinka luokan määrittely esittelee sekä luokan tietojäsenet että määrittelee luokan sisäisen esitystavan sekä luokan jäsenfunktiot, jotka määrittelevät operaatiojoukon, jota voidaan käyttää luokkatyypin olioihin. Näytämme, kuinka *tiedon piilotusta* käytetään luokan määrittelyssä luokan sisäisen esitystavan esittelyyn yksityiseksi, kun taas luokkaolioilla tehtävät operaatiot ovat julkisia. Yksityisen, sisäisen esitystavan sanotaan olevan *kapseloitu* ja luokan julkista osaa sanotaan *luokan rajapinnaksi*.

Sitten luvussa katsotaan erityisiä luokkajäseniä: *staattisia jäseniä*. Sen jälkeen katsomme, kuinka osoittimia jäseniin voidaan käyttää luokan tietojäseniin tai jäsenfunktioihin viittaamiseen. Esittelemme myös yhdisteet, jotka ovat erityisiä luokkatyyppejä erityyppisten olioiden kerrostamiseen. Luku päättyy luokan viittausalueen käsittelyyn ja sen nimiresoluutioon. Seuraavia aiheita käsitellään uudelleen erilaisten määriteltävien luokkien kannalta: sisäkkäiset luokat, luokat nimiavaruuden jäseninä ja paikalliset luokat.

13.1 Luokan määrittely

Luokan (*class*) määrittelyssä on kaksi osaa: *luokan otsikko*, joka muodostuu class-avainsanasta ja luokan nimestä ja *luokan runko* aaltosulkuparin sisällä. Luokan määrittelyn pitää päättyä puolipisteeseen tai esittelyluetteloon. Esimerkiksi:

```
class Screen { /* ... */ };
class Screen { /* ... */ } myScreen, yourScreen;
```

Luokan rungossa esitellään luokan tietojäsenet (*data members*) ja jäsenfunktiot (*member functions*) sekä näiden luokkajäsenten käsittelytasot (*access levels*). Luokan runko määrittelee *luokan jäsenluettelon* (*class member list*).

Jokainen luokkamäärittely esittelee erilaisen luokkatyypin. Jos vaikka kahdella luokkatyypillä on täsmälleen samanlainen jäsenluettelo, ne ovat siitä huolimatta eri tyyppejä. Esimerkiksi:

```
class First {
    int memi;
    double memd;
};

class Second {
    int memi;
    double memd;
};

class First obj1;
Second obj2 = obj1; // virhe: obj1 ja obj2 ovat erityyppisiä
```

Luokan runko määrittelee viittausalueen. Luokkajäsenten esittelyt luokan rungossa esittelevät jäsenten nimet luokkansa viittausalueelle. Jos kahdessa luokassa on samannimisiä jäseniä, ei ohjelma ole virheellinen, koska jäsenet viittaavat eri olioihin. Katsomme luokan viittausaluetta tarkemmin kohdassa 13.9.

Sen jälkeen, kun luokkatyyppi on esitelty, voidaan tähän luokkatyyppiin viitata kahdella tavalla:

1. Määrittämällä avainsana `class` ja luokan nimi. Edellisessä esimerkissä `obj1`:n esittely viittaa First-luokkaan tällä tavalla.
2. Määrittämällä vain luokan nimi. Edellisessä esimerkissä `obj2`:n esittely viittaa Second-luokkaan tällä tavalla.

Molemmat luokkatyyppiin viittaamisen menetelmät ovat samanarvoisia. Ensimmäinen menetelmä on lainattu C-kielestä ja se pätee myös C++-kielessä luokkatyyppien viittaukseen esittelyissä; toinen menetelmä esiteltiin C++:ssa, jotta se helpottaisi luokkatyyppien käyttöä esittelyissä.

13.1.1 Tietojäsenet

Luokan tietojäsenet esitellään samalla tavalla kuin muuttujat. Esimerkiksi Screen-luokassa voi olla seuraavia tietojäseniä:

```
#include <string>
class Screen {
    string          _screen;   // string( _height * _width )
```

```
    string::size_type  _cursor;   // nykyinen Screen:in sijainti
    short              _height;   // Screen:in rivien lukumäärä
    short              _width;    // Screen:in sarakkeiden lukumäärä
};
```

Koska olemme päättäneet käyttää merkkijonoa Screen-luokkaolion sisäiseen esitystapaan, _screen-tietojäsen on string-tyyppinen. _cursor on indeksi string-tietotyyppiin ja se viittaa nykyiseen Screen-luokkaolion sijaintipaikkaan. Sen tyyppi on

```
string::size_type
```

joka on siirrettävä tyyppi ja voi sisältää indeksin arvon string-tietojäseneen (kohdassa 6.8 esitellään size_type).

Samoin kuin muuttujaesittelyissä, tässäkään ei ole välttämätöntä esitellä kahta short-tyyppistä jäsentä erikseen. Seuraavassa on samanarvoinen Screen-määrittely kuin edellä:

```
class Screen {
/*
 * _screen osoittaa merkkijonoa, joka on kooltaan size _height * _width;
 * _cursor osoittaa nykyisen Screen:in sijaintipaikkaan;
 * _height ja _width viittaavat rivien ja sarakkeiden lukumääriin.
 */
    string             _screen;
    string::size_type  _cursor;
    short              _height, _width;
};
```

Luokan tietojäsen voi olla minkä tyyppinen tahansa. Esimerkiksi:

```
class StackScreen {
    int topStack;
    void (*handler)();    // osoitin funktioon
    vector<Screen> stack; // luokkien vektori
};
```

Tähän asti nähdyt tietojäsenet ovat *ei-staattisia* tietojäseniä. Luokalla voi olla myös *staattisia* eli luokkakohtaisia tietojäseniä. Staattisilla tietojäsenillä on erityisiä ominaisuuksia, joita tutkimme kohdassa 13.5.

Kuten olemme nähneet, tietojäsenten esittelyt näyttävät melko samanlaisilta kuin muuttujien esittelyt lohkon tai nimiavaruuden viittausalueella. Lukuunottamatta pieniä poikkeuksia, jotka koskevat staattisia jäseniä, ei tietojäseniä kuitenkaan voida alustaa eksplisiittisesti luokan rungossa. Esimerkiksi:

```
class First {
    int    memi = 0;   // virhe
    double memd = 0.0; // virhe
};
```

Luokan tietojäsenet alustetaan käyttämällä luokan *muodostajaa* (*constructor*, myös muodostin tai konstruktori). Luokan muodostajat esiteltiin lyhyesti kohdassa 2.3. Käsittelemme muodostajat ja luokan alustamisen tarkemmin luvussa 14.

13.1.2 Jäsenfunktiot

Käyttäjät haluavat tehdä monia eri operaatioita Screen-tyyppisillä oliolla. Tarvitaan joukko kohdistimen siirto-operaatioita. Pitää olla mahdollista testata ja asettaa näytön eri osia. Käyttäjän pitää pystyä kopioimaan yksi Screen-olio toiseen. Käyttäjän tulisi myös pystyä asettamaan näytön todelliset ulottuvuudet ajonaikaisesti. Nämä operaatiot voidaan toteuttaa luokan *jäsenfunktioilla*.

Luokan jäsenfunktiot esitellään luokan rungon sisällä. Jäsenfunktion esittely muodostuu siten, että se näyttää aivan kuin funktion esittelyltä, joka esiintyy nimiavaruuden viittausalueella. (Muista, että globaali viittausalue on myös nimiavaruuden viittausalue. Kohdassa 8.2 käsitellään globaaleja funktioita. Kohdassa 8.5 käsitellään nimiavaruuksia.) Esimerkiksi:

```
class Screen {
public:
    void home();
    void move( int, int );
    char get();
    char get( int, int );
    void checkRange( int, int );
    // ...
};
```

Myös jäsenfunktion määrittely voidaan sijoittaa luokan rungon sisälle. Esimerkiksi:

```
class Screen {
public:
    // määrittelyt jäsenfunktioille home() ja get()
    void home() { _cursor = 0; }
    char get()  { return _screen[_cursor]; }
    // ...
};
```

Funktio `home()` asettaa kohdistimen näytön vasempaan yläkulmaan. Funktio `get()` palauttaa kohdistimen nykyisen sijaintipaikan merkin arvon.

Jäsenfunktiot erotellaan tavallisista funktioista seuraavien ominaisuuksien perusteella:

- Jäsenfunktiot esitellään luokkansa viittausalueella. Tämä tarkoittaa, että jäsenfunktion nimi ei ole näkyvissä luokkansa viittausalueen ulkopuolella. Jäsenfunktioon viitataan käyttämällä pistettä tai nuolta jäsenen käsittelyoperaattorina kuten seuraavassa:

```
ptrScreen->home();
myScreen.home();
```

Kohdassa 13.9 käsitellään luokan viittausalue tarkemmin.

- Jäsenfunktioilla on täysi pääsy luokan sekä julkisiin että yksityisiin jäseniin, kun taas tavallisilla funktioilla on yleensä pääsy vain luokan julkisiin jäseniin. Tietystikään yhden luokan jäsenfunktioilla ei yleensä ole pääsyä toisen luokan jäseniin.

Jäsenfunktio voi olla ylikuormitettu funktio (ylikuormitetut funktiot on esitelty luvussa 9). Kuitenkin jäsenfunktio ylikuormittaa vain oman luokkansa muita jäsenfunktioita. Jäsenfunktio ei siten voi ylikuormittaa funktioita, jotka on esitelty muissa luokissa tai nimiavaruuksien viittausalueilla. Esimerkiksi esittely get(int,int) ylikuormittaa vain get()-jäsenfunktiota, joka aiemmin esiteltiin Screen-luokassa:

```
class Screen {
public:
    // ylikuormitetun get()-jäsenfunktion esittely
    char get() { return _screen[_cursor]; }
    char get( int, int );
    // ...
};
```

Katsomme luokan jäsenfunktioita tarkemmin kohdassa 13.3.

13.1.3 Jäsenten käsittely

Sattuu usein, että monet ohjelmat muokkaavat luokkatyypin sisäistä esitystapaa jälkeenpäin. Kuvitellaan esimerkiksi, että erästä tutkimusta johtavat Screen-luokkamme käyttäjät ja he ovat päättäneet, että kaikki Screen-luokasta määritellyt oliot ovat kokoa 80 x 24. Silloin on parempi toteuttaa joustamattomampi, mutta tehokkaampi esitystapa Screen-luokasta, kuten seuraavassa:

```
class Screen {
public:
    // jäsenfunktiot
private:
    // staattisen jäsenen alustaminen käsitelty kohdassa 13.5
    static const int  _height = 24;
    static const int  _width = 80;
    string            _screen;
    string::size_type _cursor;
};
```

Jäsenfunktioiden vanha toteutus — eli se, kuinka he käsittelevät luokan tietojäseniä — ei ole sopiva enää. Jäsenfunktiot pitää toteuttaa uudelleen. Tämä muutos ei kuitenkaan vaadi, että luokan jäsenfunktioiden rajapintaa (niiden parametriluetteloita ja paluutyyppejä) muutetaan.

Jos tietojäsenet olisivat julkisia ja niitä pystyisi käsittelemään ohjelman mistä tahansa funktiosta, mitä vaikutuksia Screen-luokan käyttäjille tästä luokan sisäisen esitystavan muutoksesta olisi?

- Jokainen funktio, joka suoraan käsittelee vanhan Screen-esitystavan tietojäseniä, lakkaisi toimimasta. On tarpeen paikallistaa ja kirjoittaa kaikki nuo koodiosat uudelleen ennen kuin ohjelmaa voidaan käyttää jälleen.
- Koska jäsenfunktioiden rajapinta ei ole muuttunut, yksikään funktio, joka käsittelee Screen-olioita vain Screen-jäsenfunktioiden kautta, ei vaadi muutoksia toimivaan koodiinsa. Koska kuitenkin itse jäsenfunktiot toteutetaan uudelleen, on ohjelman uudelleenkääntäminen välttämätöntä.

Tiedon piilotus on mekanismi, jolla estetään ohjelman funktioita käsittelemästä suoraan luokkatyypin sisäistä esitystapaa. Käsittelyrajoitukset luokan jäseniin määritetään otsikoilla *public*, *private* ja *protected* luokan rungossa. Avainsanoja public, private ja protected kutsutaan *käsittelymääreiksi*. Jäsenet, jotka on esitelty public-osassa, ovat julkisia jäseniä; ne, jotka on esitelty private- tai protected-osassa, ovat yksityisiä tai suojattuja jäseniä.

- *Julkiseen jäseneen* päästään ohjelmassa mistä tahansa. Luokka, joka pakottaa tiedon piilotukseen, rajoittaa julkiset jäsenensä jäsenfunktioihin, jotka määrittelevät operaatiot, joita yleiset ohjelmat voivat käyttää tuon luokkatyypin olioiden käsittelyyn.
- *Yksityiseen jäseneen* päästään vain jäsenfunktioista ja luokan *ystävistä*. Luokka, joka pakottaa tiedon piilotukseen, esittelee tietojäsenensä yksityisinä.
- *Suojattu jäsen* käyttäytyy julkisena jäsenenä *johdettuun luokkaan* ja kuin yksityinen jäsen ohjelman loppuun saakka. (Näimme luvussa 2, kuinka suojattuja jäseniä käytetään IntArray-luokassa. Suojattujen jäsenten käsittelyä pidätellään lukuun 17 saakka, jossa esitellään johdetut luokat ja käsite *periytyminen*.)

Seuraava Screen-luokan määrittely määrittää julkiset ja yksityiset osansa:

```
class Screen {
public:
    void home(){ _cursor = 0; }
    char get() { return _screen[_cursor]; }
    char get( int, int );
    void move( int, int );
    // ...
private:
    string          _screen;
    string::size_type  _cursor;
    short           _height, _width;
};
```

Tapana on, että ensiksi esitellään luokan julkiset jäsenet. (Jos haluat tietää, miksi vanhemmassa C++-koodissa esiteltiin yksityiset jäsenet ensin ja miksi sellaista tyyliä yhä esiintyy, katso julkaisua [LIPPMAN96a].) Yksityiset jäsenet ovat lueteltuna luokan rungon loppuosassa.

Luokka voi sisältää useita public-, protected- tai private-otsikoita. Jokainen osa jää voimaan, kunnes seuraava otsikko tai luokan rungon sulkeva oikeanpuoleinen aaltosulku tulee kohdalle. Ellei käsittelymäärettä ole annettu, oletusarvo on, että luokan rungon avaavan vasemmanpuoleisen aaltosulun jälkeen alkava osa on yksityinen.

13.1.4 Ystävät

Joissakin tilanteissa on kätevää sallia tiettyjen funktioiden pääsy luokan yksityisiin jäseniin sallimatta sitä kuitenkaan koko ohjelmalle. Mekanismi nimeltään *ystävä* (*friend*) mahdollistaa, että luokka voi myöntää funktioille pääsyn sen yksityisiin jäseniin.

Ystävän esittely alkaa avainsanalla friend. Se voi esiintyä vain luokan määrittelyssä. Koska ystävät eivät ole sen luokan jäseniä, joka ystävyyden myöntää, ei niihin vaikuta public-, private- tai protected-osa, jossa ne on esitelty luokan rungossa. Tässä olemme päättäneet ryhmittää kaikki ystäväesittelyt välittömästi luokan otsikon jälkeen:

```
class Screen {
    friend istream&
        operator>>( istream&, Screen& );
    friend ostream&
        operator<<( ostream&, const Screen& );
public:
    // ... loput Screen-luokasta
};
```

Syöttö- ja tulostusoperaattorit voivat nyt viitata suoraan Screen-luokan jäseniin ilman virhettä. Syöttöoperaattorin yksinkertainen toteutus voisi olla kuten seuraavassa:

```
#include <iostream>
ostream& operator<<( ostream& os, const Screen& s )
{
    // on ok viitata näihin: _height, _width ja _screen
    os << "<" << s._height
       << "," << s._width << ">";

    os << s._screen;

    return os;
}
```

Ystävä voi olla nimiavaruuden funktio, toisen, aikaisemmin määritellyn luokan jäsenfunktio tai kokonainen luokka. Kun koko luokasta tehdään ystävä, annetaan kaikille ystäväluokan jäsenfunktioille pääsy ystävyyden myöntävän luokan yksityisiin jäseniin. (Kohdassa 15.2 käsitellään ystävät yksityiskohtaisesti.)

13.1.5 Luokan esittely vastaan luokan määrittely

Luokan sanotaan olevan *määritelty* heti, kun luokan rungon loppu (tarkoittaa sulkevaa aaltosulkua) kohdataan. Kun luokka on määritelty, ovat kaikki luokan jäsenet tunnettuja. Myös luokan koko tiedetään silloin.

On mahdollista esitellä luokka määrittelemättä sitä. Esimerkiksi:

```
class Screen; // Screen-luokan esittely
```

Tämä esittelee ohjelmalle nimen Screen ja ilmaisee, että Screen viittaa luokkatyyppiin.

Luokkatyyppiä, joka on esitelty, mutta ei määritelty, voidaan käyttää vain rajoitetuin tavoin. Luokkatyypin olioita ei voida määritellä, ellei luokkaa ole määritelty, koska luokkatyypin kokoa ei tiedetä eikä kääntäjä tiedä, kuinka paljon muistia tuon tyyppiselle luokan oliolle pitäisi varata.

Kuitenkin voidaan esitellä osoitin tai viittaus tuohon luokkatyyppiin. Osoittimet ja viittaukset ovat sallittuja, koska molemmilla on kiinteä koko, joka on riippumaton sen tyypin koosta, johon se viittaa. Koska kuitenkin luokan koko ja jäsenfunktiot ovat tuntemattomia, ei käänteisoperaattoria (*) voida käyttää sellaiseen osoittimeen eikä osoitinta tai viittausta käyttää luokan jäseneen viittaamiseen ennen kuin luokka on täysin määritelty.

Tietojäsen voidaan esitellä luokan tyypiksi vain, jos luokan määrittely on jo nähty. Siellä, missä luokan määrittelyä ei vielä ole nähty ohjelman tekstitiedostossa, voi tietojäsen olla vain osoitin tai viittaus tuohon luokkatyyppiin. Tässä on esimerkiksi StackScreen-luokan tietojäsenen määrittely, joka on osoitin Screen-luokkaan, joka on esitelty, mutta ei määritelty:

```
class Screen; // esittely
class StackScreen {
    int topStack;
    // ok: osoitin Screen-olioon
    Screen *stack;
    void (*handler)();
};
```

Koska luokan ei katsota olevan määritelty ennen kuin sen runko on täydellinen, ei luokalla voi olla oman tyyppinsä mukaisia tietojäseniä. Luokan katsotaan kuitenkin olevan esitelty heti, kun luokan otsikko on nähty. Luokalla voi silloin olla tietojäseniä, jotka ovat osoittimia ja viittauksia omaan luokkatyyppiinsä. Esimerkiksi:

```
class LinkScreen {
    Screen window;
    LinkScreen *next;
    LinkScreen *prev;
};
```

Harjoitus 13.1

Olkoon luokka nimeltään Person, jossa on seuraavat tietojäsenet

```
string _name;
string _address;
```

ja seuraavat jäsenfunktiot

```
Person( const string &n, const string &a )
    : _name( n ), _address( a ) { }
string name() { return _name; }
string address() { return _address; }
```

Mitkä jäsenet esittelisit luokan julkiseen osaan ja mitkä luokan yksityiseen osaan? Perustele valintasi.

Harjoitus 13.2

Selitä ero luokan määrittelyn ja esittelyn välillä. Milloin käyttäisit luokan esittelyä? Entä luokan määrittelyä?

13.2 Luokkaoliot

Luokan kuten Screen määrittely ei aiheuta muistin varausta. Muistia varataan vain silloin, kun luokkatyypistä määritellään olio. Olkoon esimerkiksi seuraava Screen-luokan toteutus

```
class Screen {
public:
    // jäsenfunktiot
private:
    string          _screen;
    string::size_type  _cursor;
    short           _height;
    short           _width;
};
```

Tällöin määrittely

```
Screen myScreen;
```

varaa muistialuetta riittävästi, jotta siihen mahtuisi neljä Screen-luokan tietojäsentä. Nimi myScreen viittaa tuohon muistialueeseen. Jokaisella luokan oliolla eli *luokkaoliolla* (*class object*) on oma kopionsa luokan tietojäsenistä. Kun muokataan myScreen:in tietojäseniä, se ei vaikuta minkään toisen Screen-olion tietojäseniin.

Luokkatyypin olion viittausalue päätellään paikasta, jossa olion määrittely tapahtuu tekstitiedostossa. Luokkatyypin olio voidaan määritellä eri viittausalueella kuin missä luokkatyyppi on määritelty. Esimerkiksi:

```
class Screen {
    // jäsenluettelo
};

int main()
{
    Screen mainScreen;
}
```

Luokkatyyppi Screen on esitelty globaalilla viittausalueella, kun taas mainScreen-olio on esitelty main()-funktion paikallisella viittausalueella.

Luokkatyypin oliolla on myös elinaika. Riippuen siitä, onko olio esitelty nimiavaruuden viittausalueella vai paikallisella viittausalueella ja onko sitä esitelty staattiseksi vai ei, olio voi olla olemassa ohjelman koko keston ajan tai vain tietyn funktion käynnistyksen ajan. Luokkatyypin oliot käyttäytyvät melko samalla tavalla kuin muutkin oliot, kun ajatellaan niiden viittausaluetta ja elinaikaa. Olioiden viittausalue ja elinaika on esitelty luvussa 8.

Olioita, jotka ovat samaa luokkatyyppiä, voidaan alustaa ja sijoittaa toisiinsa. Oletusarvo on, että luokkaolion kopiointi on yhtä kuin kaikkien sen tietojäsenten kopiointi. Esimerkiksi:

```
Screen bufScreen = myScreen;
// bufScreen._height = myScreen._height
// bufScreen._width  = myScreen._width
// bufScreen._cursor = myScreen._cursor
// bufScreen._screen = myScreen._screen
```

Luokkaolioihin voidaan esitellä myös osoittimia ja viittauksia. Osoitin luokkatyyppiin voidaan alustaa, tai siihen voidaan sijoittaa saman luokkatyypin olion osoite. Samalla tavalla viittaus luokkatyyppiin voidaan alustaa saman luokkatyypin olion *lvaluella*. (Oliosuuntautunut ohjelmointi laajentaa tätä ja sallii kantaluokan osoittimen tai viittauksen viitata johdetun luokan olioon):

```
int main()
{
    Screen myScreen, bufScreen[10];
    Screen *ptr = new Screen;
    myScreen = *ptr;
    delete ptr;
    ptr = bufScreen;

    Screen &ref = *ptr;
    Screen &ref2 = bufScreen[6];
}
```

Oletusarvo on, että luokkaolio välitetään arvona, kun se määritetään funktion argumenttina tai funktion paluuarvona. On mahdollista esitellä funktion parametri tai paluutyyppi osoittimena tai viittauksena luokkatyyppiin. Kohdassa 7.3 esitellään parametreja, jotka ovat osoittimia tai viittauksia luokkatyyppeihin ja selitetään, miksi niitä tulisi käyttää. Kohdassa 7.4 esitellään paluutyyppejä, jotka ovat osoittimia tai viittauksia luokkatyyppeihin ja kerrotaan, miksi niitä tulisi käyttää.

Jäsenen käsittelyoperaattoreita pitää käyttää luokan joko tietojäsenten tai jäsenfunktioiden käsittelyyn. Jäsenen käsittelyn pisteoperaattoria (.) käytetään luokkaolion tai viittauksen yhteydessä; nuolioperaattoria (->) käytetään, kun luokkaoliota käsitellään osoittimella. Esimerkiksi:

```
#include "Screen.h"

bool isEqual( Screen& s1, Screen *s2 )
{ // palauta arvo false, ellei ole yhtäsuuri; ja true, jos on

    if ( s1.height() != s2->height() ||
         s1.width() != s2->width() )
             return false;

    for ( int ix = 0; ix < s1.height(); ++ix )
        for ( int jy = 0; jy < s2->width(); ++jy )
            if ( s1.get( ix, jy ) != s2->get( ix, jy ) )
                return false;

    return true; // yhä täällä? yhtäsuuri.
}
```

`isEqual()` on funktio (ei jäsenfunktio), joka vertailee kahden Screen-olion yhtäsuuruutta. `isEqual()`-funktiolla ei ole pääsyoikeutta Screen-luokan yksityisiin tietojäseniin eikä se siten voi viitata jäseniin `s1` ja `s2` suoraan. Sen täytyy luottaa Screen-luokan julkisiin jäsenfunktioihin.

Jotta `isEqual()` voisi saada arvot näytön korkeudesta ja leveydestä, pitää sen käyttää jäsenfunktioita `height()` ja `width()`, joita sanotaan *käsittelyfunktioiksi* (*access functions*). Näillä funktioilla päästään vain lukemaan luokan yksityisiä tietojäseniä. Niiden toteutus on yksinkertainen:

```
class Screen {
public:
    int height() { return _height; }
    int width()  { return _width; }
    // ...
private:
    short _height, _width;
    // ...
};
```

Kun jäsenen käsittelyssä käytetään nuolioperaattoria osoittimena luokkaolioon, se on sama, kuin käytettäisiin osoittimen käänteisoperaattoria (*) pääsemiseksi luokkaolioon, johon osoitin viittaa, ja käytettäisiin pisteoperaattoria haluttuun luokkajäseneen. Esimerkiksi lauseke

```
s2->height()
```

voidaan kirjoittaa myös näin

```
(*s2).height()
```

ja sen tulos on täsmälleen sama.

13.3 Luokan jäsenfunktiot

Luokan jäsenfunktiot toteuttavat joukon operaatioita, joita luokkaoliolla voidaan suorittaa. Operaatiojoukko, jota Screen-luokkaoliolla voidaan suorittaa, on määritelty jäsenfunktioiksi, jotka on esitelty Screen-luokassamme:

```
class Screen {
public:
    void home() { _cursor = 0; }
    void move( int, int );
    char get() { return _screen[_cursor]; }
    char get( int, int );
    bool checkRange( int, int );
    int height() { return _height; }
    int width() { return _width; }
    // ...
};
```

Vaikka jokaisella luokkaoliolla on oma kopionsa luokan tietojäsenistä, on olemassa vain yksi kopio jokaisesta luokan jäsenfunktiosta. Esimerkiksi:

```
Screen myScreen, groupScreen;
myScreen.home();
groupScreen.home();
```

Kun home()-funktiota kutsutaan myScreen-oliolle, on home():n käsittelemä _cursor-tietojäsen myScreen-olion tietojäsen. Kun home()-funktiota kutsutaan groupScreen-oliolle, viittaa _cursor-tietojäsen groupScreen-olion tietojäseneen; kuitenkin kutsutaan samaa home()-funktiota. Kuinka voi sama jäsenfunktio viitata kahden eri luokan olioiden tietojäseniin? Tämä tuki on toteutettu this-osoittimen kautta, jota tutkitaan seuraavassa kohdassa.

13.3.1 Välittömät jäsenfunktiot vastaan muut jäsenfunktiot

Huomaa, että funktioiden home(), get(), height() ja width() määrittelyt on tehty luokan runkoon. Näiden funktioiden sanotaan olevan määritelty *välittömiksi* (*inline*) luokkamäärittelyssä. Nämä funktiot käsitellään *automaattisesti* välittöminä funktioina. Välittömät funktiot on esitelty kohdassa 7.6.

Nämä jäsenfunktiot olisi myös voitu esitellä luokan rungossa välittömiksi eksplisiittisesti määrittämällä inline-avainsana ennen jäsenfunktioiden määrittelyjen paluutyyppejä kuten seuraavassa:

```
class Screen {
public:
    // Käytetään inline-avainsanaa,
    // jotta saadaan jäsenfunktiot välittömiksi
    inline void home() { _cursor = 0; }
    inline char get() { return _screen[_cursor]; }
    // ...
};
```

Tämän esimerkin home()- ja get()-määrittelyillä on täsmälleen sama merkitys kuin home()- ja get()-määrittelyillä edellisessä esimerkissä, jossa inline-avainsana oli jätetty pois. Koska se on tarpeeton, ei esimerkeissämme määritetä eksplisiittisesti inline-avainsanaa jäsenfunktioille, jotka on määritelty luokan rungossa.

Jäsenfunktiot, jotka ovat suurempia kuin yksi tai kaksi riviä, on parasta määritellä luokan rungon ulkopuolelle. Tämä vaatii erityistä esittelysyntaksia, joka yksilöi funktion luokkansa jäseneksi: jäsenfunktion nimi pitää *tarkentaa* luokkansa nimellä. Tässä on esimerkiksi checkRange()-funktion määrittely, jossa funktion nimi on tarkennettu Screen-luokalla:

```
#include <iostream>
#include "Screen.h"

// jäsenfunktion nimi on tarkennettu näin: Screen::
bool Screen::checkRange( int row, int col )
{ // tarkista koordinaatit
    if ( row < 1 || row > _height ||
        col < 1 || col > _width ) {
          cerr << "Screen coordinates ( "
              << row << ", " << col
              << " ) out of bounds.\n";
          return false;
    }
    return true;
}
```

Jäsenfunktio pitää ensin esitellä luokan rungossa ja luokan rungon on oltava näkyvissä ennen kuin jäsenfunktio voidaan määritellä luokkansa rungon ulkopuolella. Jos esimerkiksi Screen.h-otsikkotiedostoa ei olisi otettu mukaan ennen checkRange()-funktion määrittelyä, olisi edellinen

ohjelma ollut virheellinen. Luokan rungossa määritellään luokan jäsenten täydellinen luettelo. Tätä luetteloa ei voi laajentaa sen jälkeen, kun luokan runko on päättynyt.

Jäsenfunktio, joka on määritelty luokkansa rungon ulkopuolelle, ei tavallisesti ole välitön funktio. Sellainen funktio voidaan kuitenkin esitellä välittömäksi joko käyttämällä funktion esittelyssä eksplisiittistä inline-avainsanaa, joka esiintyy luokan rungossa, käyttämällä funktion määrittelyssä eksplisiittisesti inline-avainsanaa, joka esiintyy luokan rungon ulkopuolella tai käyttämällä näitä molempia. Esimerkiksi seuraava toteutus määrittelee move():n Screen-luokan välittömäksi jäsenfunktioksi:

```
inline void Screen::move( int r, int c )
{ // siirrä _cursor absoluuttiseen paikkaan
    if ( checkRange( r, c ) ) // kelvollinen näytön positio
    {
        int row = (r-1) * _width; // rivin sijaintipaikka
        _cursor = row + c - 1;
    }
}
```

Funktio get(int,int) voidaan esitellä välittömäksi määrittämällä inline-avainsana seuraavasti:

```
class Screen {
public:
    inline char get( int, int );
    // muut jäsenfunktioiden esittelyt jäävät ennalleen
};
```

Funktion määrittely tulee luokan määrittelyn jälkeen. inline-avainsana voidaan jättää pois tästä määrittelystä:

```
char Screen::get( int r, int c )
{
    move( r, c ); // aseta _cursor
    return get(); // toinen get()-jäsenfunktio
}
```

Koska välittömät funktiot pitää määritellä kaikissa tekstitiedostossa, joissa niitä kutsutaan, välitön jäsenfunktio, jota ei ole määritelty luokan rungossa, pitää sijoittaa otsikkotiedostoon, jossa luokan määrittely on. Esimerkiksi aikaisemmin esitetyt move()- ja get()-määrittelyt tulisi sijoittaa Screen.h-otsikkotiedostoon Screen-luokan määrittelyn jälkeen.

13.3.2 Pääsy luokan jäseniin

Jäsenfunktion määrittelyn sanotaan olevan luokan viittausalueella huolimatta siitä, sijaitseeko määrittely luokan rungon sisäpuolella vai ulkopuolella. Tästä seuraa kaksi asiaa:

1. Jäsenfunktion määrittely voi viitata luokan mihin tahansa jäseneen rikkomatta luokan käsittelyrajoituksia, oli jäsen yksityinen tai julkinen.
2. Jäsenfunktio voi käsitellä luokkansa jäseniä käyttämättä piste- tai nuolioperaattoria.

Esimerkiksi:

```
#include <string>

void Screen::copy( const Screen &sobj )
{
    // jos tämä Screen-olio ja sobj ovat sama olio,
    // ei kopiointi ole välttämätöntä
    // katsomme 'this'-osoitinta kohdassa 13.4
    if ( this != &sobj )
    {
        _height = sobj._height;
        _width = sobj._width;
        _cursor = 0;

        // luo uuden merkkijonon (string);
        // sen sisältö on sama kuin sobj._screen
        _screen = sobj._screen;
    }
}
```

Huomaa, että vaikka tietojäsenet _screen, _height, _width ja _cursor ovat Screen-luokan yksityisiä jäseniä, voi copy()-jäsenfunktio viitata näihin yksityisiin jäseniin ilman, että se aiheuttaisi virheen. Jos tietojäseniä kuten _screen, _height, _width ja _cursor käytetään ilman jäsenen käsittelyoperaattoria, jäsenfunktio viittaa sen luokkaolion tietojäseneen, jolle jäsenfunktiota kutsutaan. Jos esimerkiksi copy()-funktiota on käytetty kuten seuraavassa

```
#include "Screen.h"

int main()
{
    Screen s1;
    // aseta s1:n sisältö

    Screen s2;
    s2.copy(s1);

    // ...
}
```

viittaa copy()-jäsenfunktion määrittelyssä sobj-parametri s1-olioon, joka on määritelty main()-funktiossa. Ennen jäsenen käsittelyn pisteoperaattoria mainittu s2-olio on se, josta copy()-jäsenfunktiota on kutsuttu. Tässä copy()-kutsussa ovat tietojäsenet _screen, _height, _width ja _cursor ne, joihin viitataan copy():n määrittelyssä ilman jäsenen käsittelyoperaattoria, ja jotka itse asiassa viittaavat s2-olion tietojäseniin. Katsomme seuraavassa kohdassa tarkemmin pääsyä luokan jäseniin jäsenfunktioiden määrittelyistä ja näytämme, kuinka sitä tuetaan this-osoittimen käytön avulla.

13.3.3 Yksityiset jäsenfunktiot vastaan julkiset jäsenfunktiot

Jäsenfunktio voidaan esitellä luokan rungon julkisessa, suojatussa tai yksityisessä osassa. Kuinka päätellään, missä jäsenfunktio tulisi esitellä? Julkinen jäsenfunktio määrittelee operaation, jonka luokan käyttäjä haluaa suorittaa. Julkisten jäsenfunktioiden joukko määrittelee luokan *rajapinnan* (*interface*). Esimerkiksi Screen-luokan jäsenfunktiot home(), move() ja get() määrittelevät operaatiot ohjelmille, jotka käsittelevät Screen-tyyppisiä oliota.

Koska piilotamme luokan sisäisen esitystavan luokan käyttäjiltä esittelemällä tietojäsenet yksityisiksi, pitää tehdä julkisia jäsenfunktioita, joilla Screen-olioita voidaan käsitellä. Tätä sanotaan *tiedon piilottamiseksi*. Tiedon piilotus estää sen, että käyttäjän koodi muuttaisi luokan esitystapaa.

Yhtä tärkeää on, että näin suojataan luokkaolion sisäistä tilaa ohjelman satunnaiselta muokkaamiselta. Pieni funktiojoukko tekee kaikki olion muokkaukset. Jos virhe tapahtuu, virheen etsintäalue rajoittuu tähän funktiojoukkoon ja helpottaa suuresti ylläpitoa ja ohjelman toimivuutta.

Tähän mennessä olemme nähneet vain jäsenfunktioita, jotka tukevat yksityisten tietojäsenien lukukäsittelyä. Seuraavassa on kaksi set()-funktiota, joilla käyttäjä voi muokata Screen-oliota. Aluksi nämä kaksi uutta jäsenfunktiota pitää lisätä luokan runkoon:

```
class Screen {
public:
    void set( const string &s );
    void set( char ch );
    // muut jäsenfunktioiden esittelyt säilyvät muuttumattomina
};
```

Näiden funktioiden määrittelyt ovat seuraavat:

```
void Screen::set( const string &s )
{ // kirjoita merkkijono nykyiseen _cursor-positioon

    int space = remainingSpace();
    int len = s.size();
    if ( space < len ) {
        cerr << "Screen: warning: truncation: "
             << "space: " << space
             << "string length: " << len << endl;
        len = space;
    }

    _screen.replace( _cursor, len, s );
    _cursor += len - 1;
}

void Screen::set( char ch )
{
    if ( ch == '\0' )
```

```
            cerr << "Screen: warning: "
                 << "null character (ignored).\n";
        else _screen[_cursor] = ch;
    }
```

Screen-luokkamme toteutuksen oletus on, että Screen-olio ei sisällä upotettuja null-merkkejä. Tämä on syy, miksi set() ei salli null-merkkiä kirjoitettavan näytölle.

Esitetyt funktiot ovat julkisia jäsenfunktioita. Ne voidaan käynnistää mistä tahansa ohjelman alueelta. Yksityiset jäsenfunktiot voidaan kuitenkin käynnistää vain luokan muista jäsenfunktioista (ja ystävistä). Ohjelma ei voi käynnistää niitä suoraan. Yksityiset jäsenfunktiot tukevat muita jäsenfunktioita luokka-abstraktiota toteuttamalla. Funktio remainingSpace(), jota käytetään funktiossa set(const string&), on eräs näistä funktioista. Jäsenfunktio remainingSpace() on Screen-luokan yksityinen jäsenfunktio:

```
class Screen {
public:
    // muiden jäsenfunktioiden esittelyt säilyvät muuttumattomina
private:
    inline int remainingSpace();
};
```

remainingSpace() palauttaa tilan määrän, joka jää näytölle jäljelle:

```
inline int Screen::remainingSpace()
{ // nykyinen positio ei jää jäljelle
    int sz = _width * _height;
    return( sz - _cursor );
}
```

Suojattujen jäsenten täydellinen käsittely jätetään lukuun 17.

Seuraavassa on pieni ohjelma, jossa kokeillaan tähän saakka toteutettuja jäsenfunktioita:

```
#include "Screen.h"
#include <iostream>

int main() {
    Screen sobj(3,3); // muodostaja määritelty kohdassa 13.3.4
    string init("abcdefghi");

    cout << "Screen Object ("
         << sobj.height() << ", "
         << sobj.width() << " )\n\n";

    // aseta näytön sisältö
    string::size_type initpos = 0;
    for ( int ix = 1; ix <= sobj.width(); ++ix )
        for ( int iy = 1; iy <= sobj.height(); ++iy )
        {
            sobj.move( ix, iy );
            sobj.set( init[ initpos++ ] );
```

```
        }

    // tulosta näytön sisältö
    for ( int ix = 1; ix <= sobj.width(); ++ix )
    {
        for ( int iy = 1; iy <= sobj.height(); ++iy )
            cout << sobj.get( ix, iy );
        cout << "\n";
    }

    return 0;
}
```

Kun ohjelma käännetään ja suoritetaan, se generoi seuraavan tulostuksen:

```
Screen Object ( 3, 3 )

abc
def
ghi
```

13.3.4 Erityiset jäsenfunktiot

Eräs erityinen jäsenfunktiojoukko hallitsee luokkaolioita ja käsittelytoimia kuten alustusta, sijoitusta, muistinhallintaa, tyyppikonversiota ja tuhoamista. Kääntäjä käynnistää nämä funktiot usein implisiittisesti.

Alustavaa jäsenfunktiota sanotaan *muodostajaksi* (*constructor, myös muodostin tai konstruktori*). Se käynnistetään implisiittisesti joka kerta, kun olio määritellään tai varataan new-lausekkeella. Muodostaja esitellään antamalla sille luokan nimi. Seuraavassa on Screen-luokan muodostajan esittely, jossa on oletusargumenttien arvot parametreille hi, wid ja bkground:

```
class Screen {
public:
    Screen( int hi = 8, int wid = 40, char bkground = '#');
    // muut jäsenfunktioiden esittelyt jäävät muutoksitta
};
```

Tässä on Screen-muodostajan määrittely:

```
Screen::Screen( int hi, int wid, char bk ) :
    _height( hi ),  // alusta _height arvolla hi
    _width( wid ),  // alusta _width arvolla wid
    _cursor ( 0 ),  // alusta _cursor arvolla 0
    _screen( hi * wid, bk ) // _screen:in koko on hi * wid
                    // kaikki positiot alustettu
                    // merkin arvolla bk
{ // kaikki työ tehty jäsenen alustusluettelolla
  // kohdassa 14.5 käsitellään jäsenen alustusluetteloa
}
```

Screen-muodostaja alustaa automaattisesti jokaisen esitellyn Screen-olion. Esimerkiksi:

```
Screen s1;                  // Screen(8,40,'#')
Screen *ps = new Screen( 20 );  // Screen(20,40,'#')

int main() {
    Screen s(24,80,'*');        // Screen(24,80,'*')
    // ...
}
```

Luvussa 14 esitellään muodostajat, tuhoajat ja sijoitusoperaattorit tarkemmin. Luvussa 15 esitellään konversiofunktiot ja muistinhallintafunktiot yksityiskohtaisemmin.

13.3.5 Jäsenfunktiot, jolla on const- tai volatile-määre

Tavallisesti kaikki yritykset muuttaa const-oliota ohjelmassa saavat aikaan käännöksenaikaisen virheen. Esimerkiksi:

```
const char blank = ' ';
blank = '\n'; // virhe
```

Luokkaoliota ei kuitenkaan tavallisesti muokata ohjelmasta suoraan. Sen sijaan käynnistetään julkisia jäsenfunktioita, kun luokan oliota pitää muokata. Jotta kääntäjä voi ottaa huomioon luokkaolion const-tyyppisyyden, sen pitää erottaa turvalliset ja epäluotettavat jäsenfunktiot toisistaan (tarkoittaa jäsenfunktioita, jotka yrittävät muokata luokkaoliota ja niitä, jotka eivät yritä). Esimerkiksi:

```
const Screen blankScreen;
blankScreen.display();  // lukee luokan oliota
blankScreen.set( '*' ); // virhe: muokkaa luokan oliota
```

Luokan suunnittelija ilmaisee, mitkä jäsenfunktiot eivät muokkaa luokkaoliota, esittelemällä ne const-jäsenfunktioiksi. Esimerkiksi:

```
class Screen {
public:
    char get() const { return _screen[_cursor]; }
    // ...
};
```

Vain sellaiset jäsenfunktiot, jotka on esitelty const-tyyppisinä, voidaan käynnistää const-tyyppiselle luokkaoliolle. Avainsana const sijoitetaan jäsenfunktion parametriluettelon ja rungon väliin. const-tyyppiseen jäsenfunktioon, joka on määritelty luokan rungon ulkopuolella, pitää määrittää const-avainsana sekä sen esittelyyn että määrittelyyn. Esimerkiksi:

```
class Screen {
public:
    bool isEqual( char ch ) const;
    // ...
private:
    string::size_type  _cursor;
```

```
    string          _screen;
    // ...
};

bool Screen::isEqual( char ch ) const
{
    return ch == _screen[_cursor];
}
```

Ei ole sallittua esitellä const-tyyppistä jäsenfunktiota, joka muokkaa luokan tietojäsentä. Esimerkiksi seuraavassa yksinkertaistetussa Screen-määrittelyssä

```
class Screen {
public:
    int ok() const { return _cursor; }
    void error( int ival ) const { _cursor = ival; }
    // ...
private:
    string::size_type _cursor;
    // ...
};
```

ok() on sallittu const-tyyppisen jäsenfunktion määrittely, koska se ei muuta _cursor-arvoa. Kuitenkin error()-määrittely muokkaa _cursor-arvoa, eikä sitä siksi voi esitellä const-tyyppisenä jäsenfunktiona. Funktion määrittely johtaa seuraavaan kääntäjän virheilmoitukseen:

```
error: cannot modify a data member within a const member function
```

Yleensä kaikkien luokkien, joita otaksutaan käytettävän paljon, tulisi esitellä sellaiset jäsenfunktionsa const-tyyppisinä, jotka eivät muokkaa luokan tietojäseniä. Kuitenkaan jäsenfunktion esittely const-tyyppisenä ei estä kaikkia muokkauksia, joita ohjelmoija voi olettaa. Jäsenfunktion esitteleminen const-tyyppiseksi takaa sen, että jäsenfunktio ei muokkaa luokan tietojäseniä. Mutta jos luokka sisältää osoittimia, voidaan olioita, joihin osoittimet viittaavat, muokata const-tyyppisestä jäsenfunktiosta. Tätä muokkausta ei kääntäjä havaitse. Tämä saa aloittelevat C++-ohjelmoijat usein ymmälle. Esimerkiksi:

```
#include <cstring>

class Text {
public:
    void bad( const string &parm ) const;
private:
    char *_text;
};

void Text::bad( const string &parm ) const
{
    _text = parm.c_str(); // virhe: _text ei ole muokattavissa

    for ( int ix = 0; ix < parm.size(); ++ix )
```

```
        _text[ix] = parm[ix]; // huono tyyli, mutta ei ole virhe
}
```

Vaikka _text ei ole muokattavissa, se on kuitenkin char*-tyyppinen, ja merkit, joihin _text viittaa, ovat muokattavissa Text-luokan const-tyyppisestä jäsenfunktiosta. Jäsenfunktio bad() on esimerkki huonosta ohjelmointityylistä. Kääntäjä ei kuitenkaan auta havaitsemaan tuollaisia tilanteita ja ohjelmoijan pitää olla valppaana, koska const-tyyppinen jäsenfunktio ei takaa, että kaikki, mihin luokkaolio viittaa, säilyy muuttumattomana jäsenfunktion käynnistyksen ajan.

const-tyyppinen jäsenfunktio voidaan ylikuormittaa const-tyypittömällä jäsenfunktiolla, jolla on samanlainen parametriluettelo. Esimerkiksi:

```
class Screen {
public:
    char get(int x, int y);
    char get(int x, int y) const;
    // ...
};
```

Tässä tapauksesa luokkaolion const-tyypittömyys määrää, kumpi kahdesta funktiosta käynnistetään:

```
int main() {
    const Screen cs;
    Screen s;

    char ch = cs.get(0,0); // kutsuu const-tyyppistä jäsentä
    ch = s.get(0,0);       // kutsuu const-tyypitöntä jäsentä
}
```

Muodostajat ja tuhoajat ovat poikkeuksia siinä, että vaikka ne eivät koskaan ole const-tyyppisiä jäsenfunktioita, niitä voidaan kutsua const-tyyppisille luokkaolioille. Luokkaolion const-tyyppisyys vahvistetaan, kun muodostaja saa päätökseen suorituksensa ja luokkaolio on alustettu. Olion const-tyyppisyys häviää, kun tuhoaja on käynnistetty. Luokan const-tyyppistä oliota pidetään siten const-tyyppisenä sen muodostajan päättymishetkestä sen tuhoajan alkamishetkeen.

Jäsenfunktio voidaan esitellä myös volatile-määreellä (volatile-määre esiteltiin kohdassa 3.13). Luokkaolio esitellään volatile:ksi, jos sen arvoa voidaan mahdollisesti muuttaa jollain tavalla kääntäjän kontrollin ulkopuolella (jos se on esimerkiksi tietorakenne, joka edustaa I/O-porttia). Samoin kuin const-luokkaoliot, vain volatile-jäsenfunktiot, muodostajat ja tuhoajat voidaan käynnistää volatile-luokkaoliolle:

```
class Screen {
public:
    char poll() volatile;
    // ...
};
char Screen::poll() volatile { ... }
```

13.3.6 Muuttuvat tietojäsenet

Saattaa kuitenkin aiheutua ongelmia, kun esittelemme Screen-luokastamme olion const-tyyppisenä. Olettamuksemme on, että kun const-tyyppinen Screen-olio on alustettu, sen sisältöä ei voi muokata. Mutta Screen-olion sisältö tulisi pystyä tutkimaan ongelmitta. Jos esimerkiksi on olemassa seuraava const-tyyppinen Screen-luokan cs-olio

```
const Screen cs( 5, 5 );
```

ja haluamme tutkia cs:n sisällön paikassa (3, 4), teemme sen seuraavasti:

```
// lue Screen positiosta ( 3, 4 )
// hups: ei toimi!
cs.move( 3, 4 );
char ch = cs.get();
```

Tämä ei kuitenkaan toimi. Huomaatko, miksi? move() ei ole const-tyyppinen jäsenfunktio eikä siitä voi tehdä sellaista helposti. move():n määrittely on seuraavanlainen:

```
inline void Screen::move( int r, int c )
{
    if ( checkRange( r, c ) )
    {
        int row = (r-1) * _width;
        _cursor = row + c - 1; // muokkaa _cursor:ia
    }
}
```

Huomaa, että move() muokkaa luokan _cursor-tietojäsentä eikä sitä siitä syystä voi esitellä const-tyyppiseksi jäsenfunktioksi ilman muokkausta.

Voi kuitenkin näyttää oudolta, että _cursor:ia ei voi muokata Screen-luokan const-tyyppiselle oliolle. _cursor on vain indeksi. Kun muokkaamme _cursor:ia, emme muokkaa itse Screen:in sisältöä. Yritämme vain muistaa tutkia Screen:in position. _cursor:in muokkaus tulisi sallia, vaikka Screen-olio on const-tyyppinen, koska niin voidaan tutkia Screen-olion sisältö eikä muuteta itse Screen:in sisältöä.

Jotta voisimme sallia luokan tietojäsenen muokkauksen, vaikka se on const-tyyppisen olion tietojäsen, voimme esitellä jäsenen *muuttuvaksi* (*mutable*). Muuttuva tietojäsen on sellainen, joka ei koskaan ole const, vaikka se on const-tyyppisen olion tietojäsen. Muuttuvaa jäsentä voidaan aina päivittää, jopa const-tyyppisessä jäsenfunktiossa. Jotta jäsen voidaan esitellä muuttuvaksi tietojäseneksi, pitää laittaa mutable-avainsana ennen tietojäsenen esittelyä luokan jäsenluettelossa:

```
class Screen {
public:
    // jäsenfunktiot
private:
    string                  _screen;
    mutable string::size_type  _cursor; // muuttuva jäsen
    short                   _height;
```

```
    short               _width;
};
```

Mikä tahansa const-tyyppinen jäsenfunktio voi nyt muokata _cursor:ia ja voimme esitellä move()-jäsenfunktion const-tyyppisenä jäsenfunktiona. Vaikka move() muokkaa _cursor-tietojäsentä, ei siitä aiheudu käännösvirhettä:

```
// move() on const-tyyppinen jäsenfunktio
inline void Screen::move( int r, int c ) const
{
    // ...

    // ok: const-tyyppinen jäsenfunktio voi muokata muuttuvia jäseniä
    _cursor = row + c - 1;
    // ...
}
```

Operaatiot, jotka esiteltiin tämän alikohdan alussa, voidaan nyt suorittaa, ja voidan tutkia const-tyyppinen Screen-olio cs ilman virheitä.

Huomaa, että vain _cursor on esitelty muuttuvaksi tietojäseneksi. _screen, _height ja _width eivät ole, koska näiden tietojäsenten arvoja ei tulisi koskaan muuttaa Screen-luokkaoliossa, joka on const.

Harjoitus 13.3

Selitä copy():n käyttäytyminen seuraavassa käynnistyksessä:

```
Screen myScreen;
myScreen.copy( myScreen );
```

Harjoitus 13.4

Kohdistimen lisäliikkeisiin voisi kuulua siirtyminen yhden merkin taakse tai eteenpäin kerrallaan. Kun saavutetaan näytön vasen yläkulma tai oikea alakulma, kohdistin kiertyy ympäri. Toteuta funktiot forward() ja backward().

Harjoitus 13.5

Toinen hyödyllinen mahdollisuus voisi olla kohdistimen siirto yhden rivin ylös tai alas näytöllä. Kun saavutetaan näytön ylin rivi tai alin rivi, kohdistin ei kierry ympäri; se saa aikaan summerin äänen ja jää paikalleen. Toteuta funktiot up() ja down() tietäen, että kun kirjoitetaan merkki '007' cout:iin, saadaan ääni kuuluviin.

Harjoitus 13.6

Käy uudelleen läpi Screen-jäsenfunktiot, jotka on esitelty tähän saakka, ja muuta ne const-tyyppisiksi soveltuvin osin. Perustele päätöksesi.

13.4 Implisiittinen this-osoitin

Luokan jokainen olio ylläpitää omia kopioita luokan tietojäsenistä. Esimerkiksi:

```
int main() {
    Screen myScreen( 3, 3 ), bufScreen;

    myScreen.clear();
    myScreen.move( 2, 2 );
    myScreen.set( '*' );
    myScreen.display();

    bufScreen.reSize( 5, 5 );
    bufScreen.display();
}
```

myScreen-oliolla on omat _width-, _height-, _cursor- ja _screen-tietojäsenensä. bufScreen-oliolla on omat erilliset tietojäsenet. Kuitenkin on olemassa vain yksi kopio jokaisesta luokan jäsenfunktiosta. Sekä myScreen että bufScreen kutsuvat samaa kopiota tietystä jäsenfunktiosta.

Olemme nähneet edellisessä kohdassa, että jäsenfunktio voi viitata luokkansa jäseniin käyttämättä jäsenen käsittelyoperaattoreita. Esimerkiksi funktion move() määrittely on seuraavanlainen:

```
inline void Screen::move( int r, int c )
{
    if ( checkRange( r, c ) ) // onko kelvollinen näytön positio?
    {
        int row = (r-1) * _width; // rivin sijainti
        _cursor = row + c - 1;
    }
}
```

Jos funktiota move() kutsutaan myScreen-oliolle, sen käsittelemät tietojäsenet _width ja _cursor ovat myScreen-olion tietojäseniä. Jos funktiota move() kutsutaan bufScreen-oliolle, ovat käsitellyt tietojäsenet bufScreen-olion. Kuinka move():n käsittelemä tietojäsen _cursor on vuoroin myScreen-olion tietojäsen ja vuorin bufScreen-olion? Lyhyt vastaus on this-osoitin.

Jokainen luokan jäsenfunktio sisältää osoittimen, joka osoittaa olioon, jolle jäsenfunktiota on kutsuttu. Osoittimen nimi on this. Sen tyyppi on const-tyypittömässä jäsenfunktiossa osoitin luokkatyyppiin, osoitin const-tyyppiseen luokkatyyppiin const-tyyppisessä jäsenfunktiossa ja osoitin volatile-tyyppiseen luokkatyyppiin volatile-tyyppisessä jäsenfunktiossa. Esimerkiksi Screen-luokan jäsenfunktiossa move() on this-osoitin tyyppiä Screen*. List-luokan const-tyypittömässä jäsenfunktiossa this-osoitin on tyyppiä List*.

Koska this-osoitin osoittaa luokkaoliota, jolle jäsenfunktiota on kutsuttu, niin myScreen-oliolle kutsutussa move()-funktiossa this-osoitin osoittaa myScreen-olioon. Samalla tavalla, jos move()-funktiota on kutsuttu bufScreen-oliolle, osoittaa this-osoitin bufScreen-olioon. Tietojäsen _cursor, jota move() käsittelee, tulee sidotuksi tietojäseneen, joka kuuluu vuorollaan myScreen-oli-

olle ja bufScreen-oliolle.

Eräs tapa tämän ymmärtämiseksi on tehdä pikakatsaus siitä, kuinka kääntäjä toteuttaa this-osoittimen. On olemassa kaksi muunnosta, joita pitää käyttää this-osoittimen tukemiseksi:

1. Tulkitse luokan jäsenfunktion määrittely. Jokainen luokan jäsenfunktio on määritelty yhdellä lisäparametrilla: this-osoittimella. Esimerkiksi:

```
// pseudokoodi, joka kuvaa kääntäjän laajennusta
// jäsenfunktion määrittelystä --
// ei kelvollista C++-koodia
inline void move( Screen* this, int r, int c )
{
    if ( checkRange( r, c ) )
    {
        int row = (r-1) * this->_width;
        this->_cursor = row + c - 1;
    }
}
```

   Jäsenfunktion määrittelyssä on this-osoittimen käytöstä tehty eksplisiittinen, jotta luokan tietojäseniä _width ja _cursor voidaan käsitellä.

2. Tulkitse luokan jokainen jäsenfunktion käynnistys ja lisää lisäargumentti — sen olion osoite, jolle jäsenfunktio on käynnistetty. Esimerkiksi

```
myScreen.move( 2, 2 )
```

   tulkitaan näin

```
move( &myScreen, 2, 2 )
```

Ohjelmoija voi viitata this-osoittimeen eksplisiittisesti jäsenfunktion määrittelyssä. On esimerkiksi sallittua, vaikkakaan ei tarpeellista, määritellä jäsenfunktio home() seuraavasti:

```
inline void Screen::home()
{
    this->_cursor = 0;
}
```

On kuitenkin tilanteita, jolloin ohjelmoijan pitää viitata this-osoittimeen eksplisiittisesti kuten näimme Screen-luokkaan määritellystä copy()-jäsenfunktiosta aikaisemmin. Seuraavassa alikohdassa esitetään joitakin esimerkkejä.

13.4.1 Milloin this-osoitinta tulisi käyttää?

Funktiomme main() kutsuu Screen-luokan jäsenfunktioita myScreen- ja bufScreen-olioille niin, että jokainen toimenpide on erillisessä lauseessa. Voimme määritellä Screen-luokan jäsenfunktiot niin, että ne sallivat jäsenfunktioiden kutsujen yhdistämisen, kun niitä käytetään samaan Screen-olioon. Esimerkiksi kutsut main()-funktiossa olisi voitu kirjoittaa näin

```
int main() {
    // ...
```

```
    myScreen.clear().move(2,2).set('*').display();
    bufScreen.reSize(5,5).display();
}
```

Tämä näyttää noudattavan vaistomaista tapaa Screen-olioiden käsittelylle, jossa tehdään useita toimenpiteitä: tyhjennä (clear) Screen-luokan myScreen-olio, siirrä (move) sen kohdistin paikkaan (2,2), aseta (set) tähän paikkaan merkki '*' ja näytä (display) sitten tulos.

Jäsenen käsittelyoperaattorit, piste ja nuoli, ovat vasemmalta assosiatiivisia. Kun näiden operaattoreiden jono löydetään, on suoritusjärjestys vasemmalta oikealle. Esimerkiksi myScreen.clear() käynnistetään ensin, sen jälkeen myScreen.move() jne. Jotta myScreen.move() voitaisiin käynnistää myScreen.clear()-kutsun jälkeen, pitää clear()-kutsun palauttaa myScreen-luokkaolio. Jäsenfunktion clear() määrittelyn pitää palauttaa luokkaolio, jolle se käynnistetään. Kuten olemme nähneet, luokkaolion käsittely tapahtuu luokan jäsenfunktiossa this-osoittimen kautta. Seuraavassa clear():in toteutus:

```
// clear():in esittely on luokan rungossa
// se määrittää oletusargumentin: bkground = '#'
Screen& Screen::clear( char bkground )
{ // alusta kohdistin ja tyhjennä näyttö

    _cursor = 0;
    _screen.assign(       // sijoita merkkijonoon
        _screen.size(),    // määrältään size() merkkiä
        bkground           // joiden jokaisen arvo on bkground
    );

    // palauta olio, jolle funktio käynnistettiin
    return *this;
}
```

Huomaa, että jäsenfunktion paluutyyppi on Screen&, joka ilmaisee, että jäsenfunktio palauttaa viittauksen olioon, joka on sen oman luokan tyyppinen. Jotta voisimme mahdollistaa Screen-luokan jäsenfunktioiden yhdistämisen main()-funktiossa, pitää palata move()- ja set()-jäsenfunktioihin. Niiden paluutyyppi pitää vaihtaa void-tyypistä Screen&-tyypiksi ja niiden pitää palauttaa *this määrittelyissään.

Samalla tavalla Screen-luokan jäsenfunktio display() voitaisiin toteuttaa seuraavasti:

```
Screen& Screen::display()
{
    typedef string::size_type idx_type;

    for ( idx_type ix = 0; ix < _height; ++ix )
    { // jokaiselle riville

        idx_type offset = _width * ix; // rivipositio

        for ( idx_type iy = 0; iy < _width; ++iy )
            // kirjoita elementti jokaiselle sarakkeelle
```

```
                cout << _screen[ offset + iy ];

            cout << endl;
        }
        return *this;
}
```

Screen-luokan jäsenfunktio reSize() voitaisiin toteuttaa seuraavasti:

```
// reSize():n esittely on luokan rungossa
// se määrittää oletusargumentin: bkground = '#'
Screen& Screen::reSize( int h, int w, char bkground )
{ // muuta näytön kokoa: korkeus = h ja leveys = w

    // muista näytön sisältö
    string local(_screen);

    // korvaa merkkijonon, johon _screen viittaa
    _screen.assign(     // sijoita merkkijonoon
        h * w,          //  h * w merkkiä
        bkground        // joiden jokaisen arvo on bkground
    );

    typedef string::size_type idx_type;
    idx_type local_pos = 0;

    // kopioi vanhan näytön sisältö uuteen
    for ( idx_type ix = 0; ix < _height; ++ix )
    { // jokaiselle riville

        idx_type offset = _width * ix; // rivipositio
        for ( idx_type iy = 0; iy < _width; ++iy )
            // sijoita vanha arvo jokaiseen sarakkeeseen
            _screen[ offset + iy ] = local[ local_pos++ ];
    }

    _height = h;
    _width = w;
    // _cursor säilyy muuttumattomana

    return *this;
}
```

Kaikki this-osoittimen määrittelyt jäsenfunktioissa eivät palauta oliota, johon jäsenfunktiota on käytetty. Kun esittelimme copy()-jäsenfunktion kohdassa 13.3, näimme toisen tavan käyttää this-osoitinta:

```
void Screen::copy( const Screen& sobj )
{
    // jos tämä (this) Screen-olio ja sobj ovat sama olio,
    // ei kopiointi ole tarpeen
```

```
    if ( this != &sobj )
    {
        // kopioi sobj-olion arvo *this-olioon
    }
}
```

this-osoitin sisältää luokan olion osoitteen, jolle jäsenfunktiota on kutsuttu. Jos olion osoite, johon sobj viittaa, on sama kuin this-osoittimen arvo, silloin sekä sobj että this viittaavat samaan olioon eikä kopiointioperaatio ole tarpeen. Palaamme tähän rakenteeseen jälleen, kun katsomme kopioinnin sijoitusoperaattoria kohdassa 14.7.

Harjoitus 13.7

this-osoitinta voidaan käyttää sen luokkaolion muokkaamiseen, johon se viittaa. Voidaan myös korvata olio uudella samantyyppisellä oliolla. Esimerkiksi tässä on classType-luokan assign()-jäsenfunktio. Osaatko kertoa, mitä se tekee?

```
classType& classType::assign( const classType &source )
{
    if ( this != &source )
    {
        this->~classType();
        new (this) classType( source );
    }
    return *this;
}
```

Muista, että ~classType() on tuhoajan nimi. new-lauseke voi näyttää hieman hassulta, mutta olemme nähneet kohdassa 8.4 tämän new-lausekkeen käyttötavan, jota kutsutaan new-lausekkeen asemoinniksi.

Mikä on sinun mielipiteesi tämänkaltaisesta ohjelmointityylistä? Uskotko, että tämä on turvallinen operaatio? Miksi uskot tai miksi et usko?

13.5 Staattiset luokan jäsenet

Joskus on tarpeen, että kaikki tietyntyyppisen luokan oliot pääsevät käsittelemään globaalia oliota. Ehkäpä pitää laskea, kuinka monta tietyn tyyppistä oliota on luotu ohjelman aikana tai globaali olio voi olla osoitin luokan virheenkäsittelyrutiiniin tai se voi olla osoitin tämäntyyppisten olioiden vapaavarastomuistiin. Tällaisissa tapauksissa on yksinkertaisesti tehokkaampaa tehdä yksi globaali olio, jota kaikki tietyn tyyppisen luokan oliot käyttävät sen sijaan, että jokaisella luokkaoliolla olisi oma erillinen tietojäsen. Vaikka tämä olio on globaali, se on olemassa vain sitä tarkoitusta varten, että se tukisi tämän luokka-abstraktion toteutusta.

Tällaisissa tapauksissa on luokkakohtainen eli *staattinen tietojäsen* parempi ratkaisu. Staattinen tietojäsen toimii kuin globaali olio, joka kuuluu luokkatyypilleen. Toisin kuin muut

tietojäsenet, joista jokaisella luokkaoliolla on oma kopionsa, on staattisesta tietojäsenestä vain yksi kopio per luokkatyyppi. Staattinen tietojäsen on yksittäinen, kaikkien luokkaolioiden käytettävissä oleva jaettu olio.

On olemassa kaksi etua staattisen tietojäsenen käyttämisestä globaalin olion asemesta:

1. Staattista jäsentä ei lisätä ohjelman globaaliin nimiavaruuteen, joka siten poistaa satunnaisen mahdollisuuden nimien konfliktilta ohjelmamme muiden globaalien olioiden kanssa.
2. Staattinen tietojäsen voidaan pakottaa tiedon piilotukseen. Staattinen jäsen voi olla yksityinen, globaali olio ei voi olla.

Tietojäsen tehdään staattiseksi laittamalla sen esittelyn eteen luokan rungossa avainsana static. Staattiset tietojäsenet välttävät julkinen/yksityinen/suojattu-käsittelysäännöt. Esimerkiksi seuraavassa määrittelyssä on Account-luokan _interestRate esitelty yksityisenä staattisena jäsenenä, jonka tyyppi on double.

```
class Account {
    Account( double amount, const string &owner );
    string owner() { return _owner; }
private:
    static double  _interestRate;
    double         _amount;
    string         _owner;
};
```

Miksi _interestRate on esitelty staattisena, kun taas _amount ja _owner ei? Se johtuu siitä, että jokaisella Account-oliolla on eri omistaja ja ne sisältävät eri määrän rahaa, mutta korkoprosentti on kaikilla Account-olioilla sama.

Koska on olemassa vain yksi _interestRate-tietojäsen, jonka kaikki Account-oliot jakavat keskenään koko ohjelmassa, _interestRate:n esittely staattiseksi tietojäseneksi vähentää tarvittavaa muistia jokaiselta Account-oliolta.

Vaikka _interestRate:n nykyinen arvo on sama jokaiselle Account-oliolle, voi sen arvo muuttua ajan mittaan. Sen vuoksi päätimme, että emme esittele staattista tietojäsentä const-tyyppiseksi. Koska _interestRate on staattinen, se pitää päivittää vain kerran. Olemme varmistaneet, että jokainen Account-olio tulee käsittelemään samaa päivitettyä arvoa. Jos jokainen luokkaolio ylläpitäisi omaa kopiotaan, pitäisi jokainen kopio päivittää, mikä johtaisi tehottomuuteen ja virheen mahdollisuus olisi suurempi.

Yleensä staattinen tietojäsen alustetaan luokkamäärittelyn ulkopuolella. Aivan kuten jäsenfunktioiden yhteydessä, jotka on määritelty luokkamäärittelyn ulkopuolella, pitää staattisen jäsenen nimi tarkentaa luokkansa nimellä. Tässä on esimerkki, kuinka voisimme alustaa _interestRate:n:

```
// staattisen luokkajäsenen eksplisiittinen alustus

#include "account.h"
double Account::_interestRate = 0.0589;
```

Kuten kaikista globaaleista olioista, myös staattisesta tietojäsenestä sallitaan vain yksi määrittely ohjelmaan. Tämä tarkoittaa, että staattisten tietojäsenten alustuksia ei tulisi sijoittaa otsikkotiedostoihin, vaan tiedostoihin, jotka sisältävät luokan muiden kuin välittömien jäsenfunktioiden määrittelyt.

Staattiset tietojäsenet voidaan esitellä minkä tahansa tyyppisiksi. Ne voivat olla const-olioita, taulukkoja, luokkaolioita jne. Esimerkiksi:

```
#include <string>
class Account {
    // ...
private:
    static const string name;
};

const string Account::name( "Savings Account" );
```

Staattinen const-tietojäsen, joka on kokonaistyyppinen, voidaan erikoistapauksena alustaa luokan rungon sisällä vakioarvolla. Jos esimerkiksi päättäisimme käyttää merkkitaulukkoa string-olion sijasta tilin nimelle, määrittäisimme taulukon koon int-tyyppisellä const-tietojäsenellä. Esimerkiksi:

```
// otsikkotiedosto
class Account {
    // ...
private:
    static const int nameSize = 16;
    static const char name[nameSize];
};

// tekstitiedosto
const int Account::nameSize; // jäsenen määrittely tarvitaan
const char Account::name[nameSize] = "Savings Account";
```

On muutamia mielenkiintoisia asioita, jotka kannattaa huomioida tästä erikoistilanteesta. Staattinen const-tietojäsen, joka on kokonaistyyppiä ja alustetaan vakioarvolla, on *vakiolauseke*. Luokan suunnittelija voi esitellä sellaisen staattisen tietojäsenen, jos on tarvetta käyttää nimettyä vakioarvoa luokan rungossa. Koska esimerkiksi staattinen const-tietojäsen nameSize on vakiolauseke, luokan suunnittelija käyttää sitä määrittääkseen name-tietojäsenen taulukon koon.

Kun staattinen const-tietojäsen alustetaan luokan rungossa, pitää tietojäsen silti määritellä luokan ulkopuolella. Koska kuitenkin staattisen tietojäsenen alkuarvo määritetään luokan rungossa, ei luokan ulkopuolella oleva määrittely saa määrittää alkuarvoa.

Koska name on taulukko (eikä kokonaistyyppinen), ei sitä voi alustaa luokan rungossa. Jokainen yritys tehdä niin johtaa käännöksenaikaiseen virheeseen. Esimerkiksi:

```
class Account {
    // ...
private:
    static const int nameSize = 16; // ok: kokonaistyyppi
    static const char name[nameSize] =
                "Savings Account"; // virhe
};
```

name pitää alustaa luokkamäärittelyn ulkopuolella.

Tämä esimerkki kuvastaa erästä viimeisistä asioista. Huomaa, että jäsen nameSize määrittää name-taulukon koon, joka esiintyy luokan määrittelyn ulkopuolella:

```
const char Account::name[nameSize] = "Savings Account";
```

nameSize on tarkentamatta Account-luokan nimellä. Ja vaikka nameSize on yksityinen jäsen, ei name:n määrittely ole virheellinen. Miten se voi olla mahdollista? Aivan kuten luokan jäsenfunktion määrittely voi viitata luokan yksityisiin jäseniin, niin voi tehdä myös staattisen tietojäsenenkin määrittely. Staattisen name-tietojäsenen määrittely on sen luokan viittausalueella ja voi viitata Account-luokan yksityisiin tietojäseniin sen jälkeen, kun tarkennettu Account::name on nähty. Näemme lisää luokan viittausalueesta kohdassa 13.9.

Luokan staattista tietojäsentä voidaan käsitellä luokkansa jäsenfunktiossa käyttämättä jäsenen käsittelyoperaattoria:

```
inline double Account::dailyReturn()
{
    return( _interestRate / 365 * _amount );
}
```

Kuitenkin muissa funktioissa pitää staattista tietojäsentä käsitellä kahdella vaihtoehtoisella tavalla. Jäsenen käsittelyoperaattoria voidaan käyttää näin:

```
class Account {
    // ...
private:
    friend int compareRevenue( Account& , Account* );
    // loppu säilyy muuttumattomana
};

// viittaus- ja osoitinparametrit kuvaamaan
// olio- ja osoitinkäsittelyä
int compareRevenue( Account &ac1, Account *ac2 )
{
    double ret1, ret2;
    ret1 = ac1._interestRate * ac1._amount;
    ret2 = ac2->_interestRate * ac2->_amount;
    // ...
}
```

Sekä ac1._interestRate että ac2->_interestRate viittaavat staattiseen jäseneen Account::_interestRate.

Koska luokan staattisesta tietojäsenestä on vain yksi kopio, ei sitä tarvitse käsitellä olion tai osoittimen kautta. Toinen tapa käsitellä staattista tietojäsentä on tehdä se suoraan tarkentamalla sen nimi luokan nimellä:

```
// staattista jäsentä käsitellään tarkennetulla nimellä
if ( Account::_interestRate < 0.05 )
```

Kun staattista tietojäsentä ei käsitellä luokan jäsenen käsittelyoperaattorilla, pitää määrittää luokan nimi ja viittausalueoperaattori

```
Account::
```

koska staattinen jäsen ei ole globaali olio eikä sitä löydy globaalilta viittausalueelta. Seuraava compareRevenue()-ystäväfunktion määrittely on sama asia kuin edellä esitetty:

```
int compareRevenue( Account &ac1, Account *ac2 )
{
    double ret1, ret2;
    ret1 = Account::_interestRate * ac1._amount;
    ret2 = Account::_interestRate * ac2->_amount;
    // ...
}
```

Staattisen tietojäsenen yksilöllinen luonne — että on olemassa yksittäinen, itsenäinen ilmentymä kaikille luokkaolioille — mahdollistaa sen, että sitä voidaan käyttää tavoin, jotka eivät ole sallittuja muuntyyppisille tietojäsenille:

1. Staattinen tietojäsen voi kuulua samaan luokkatyyppiin, jossa se on jäsenenä. Ei-staattisen tietojäsenen esittely on rajoitettu osoittimeksi tai viittaukseksi luokkansa olioon. Esimerkiksi:

```
class Bar {
public:
    // ...
private:
    static Bar mem1; // ok
    Bar *mem2;       // ok
    Bar mem3;        // virhe
};
```

2. Staattinen tietojäsen voi esiintyä oletusargumenttina luokan jäsenfunktiolle, mutta ei-staattinen jäsen ei voi. Esimerkiksi:

```
extern int var;

class Foo {
private:
    int var;
    static int stcvar;
public:
    // virhe: ratkaisu johtaa ei-staattiseen Foo::var:iin
```

```
    // asianmukaista luokkaoliota ei ole
    int mem1( int = var );

    // ok: ratkaisu johtaa staattiseen Foo::stcvar:iin
    // asiaan kuuluva luokan olio tarpeeton
    int mem2( int = stcvar );

    // ok: global instance of int var
    int mem3( int = ::var );
};
```

13.5.1 Staattiset jäsenfunktiot

Jäsenfunktiot raiseInterest() ja interest() käsittelevät luokan staattista tietojäsentä _interestRate:

```
class Account {
public:
    void raiseInterest( double incr );
    double interest() { return _interestRate; }
private:
    static double _interestRate;
};

inline void Account::raiseInterest( double incr )
{
    _interestRate += incr;
}
```

Ongelma on, että jokaista jäsenfunktiota pitää kutsua käyttämällä luokan jäsenen käsittely-operaattoria tietylle luokkaoliolle. Koska jäsenfunktiot eivät käsittele muita tietojäseniä kuin staattista jäsentä _interestRate, on todella yhdentekevää, mitä oliota käytetään funktioiden kutsuun. Yhtäkään olion (ei-staattista) tietojäsentä ei koskaan käsitellä tai muokata sellaisen kutsun tuloksena.

Parempi vaihtoehto on esitellä sellainen jäsenfunktio staattisena. Se voidaan tehdä seuraavasti:

```
class Account {
public:
    static void raiseInterest( double incr );
    static double interest() { return _interestRate; }
private:
    static double _interestRate;
};

inline void Account::raiseInterest( double incr )
{
    _interestRate += incr;
}
```

Staattisen jäsenfunktion esittely on samanlainen kuin ei-staattisen jäsenfunktion, paitsi että luokan rungossa ennen funktion esittelyä on avainsana static, eikä funktiota tulisi esitellä const-tai volatile-tyyppiseksi. Funktiomäärittelyyn, joka esiintyy luokan rungon ulkopuolella, ei saa määrittää static-avainsanaa.

Staattisella jäsenfunktiolla ei ole this-osoitinta. Tästä syystä, jos viitataan joko implisiittisesti tai eksplisiittisesti this-osoittimeen staattisessa jäsenfunktiossa, se johtaa käännöksenaikaiseen virheeseen. Jos yritetään käsitellä ei-staattista luokan jäsentä, se viittaa implisiittisesti this-osoittimeen ja siten johtaa käännöksenaikaiseen virheeseen. Esimerkiksi aiemmin esiteltyä jäsenfunktiota dailyReturn() ei voitaisi esitellä staattiseksi jäsenfunktioksi, koska se käsittelee ei-staattista tietojäsentä _amount.

Staattinen jäsenfunktio voidaan käynnistää luokkaoliolle tai osoittimelle luokkaolioon käyttämällä jäsenen käsittelyoperaattoria, pistettä tai nuolta. Staattista jäsenfunktiota voidaan myös käsitellä tai käynnistää suoraan käyttämällä tarkennettua nimeä, vaikka luokkaolioita ei koskaan esitellä. Tässä on pieni ohjelma, joka kuvastaa luokan staattisten jäsenten käyttöä:

```
#include <iostream>
#include "account.h"

bool limitTest( double limit )
{
    // yhtäkään Account-oliota ei ole määritelty vielä
    // ok: kutsu staattista jäsenfunktiota
    return limit <= Account::interest() ;
}

int main() {
    double limit = 0.05;

    if ( limitTest( limit ) )
    {
        // osoitin staattiseen luokan jäseneen on
        // esitelty tavallisena osoittimena
        void (*psf)(double) = &Account::raiseInterest;
        psf( 0.0025 );
    }

    Account ac1( 5000, "Asterix" );
    Account ac2( 10000, "Obelix" );
    if ( compareRevenue( ac1, ac2 ) > 0 )
        cout << ac1.owner()
            << " is richer than "
            << ac2.owner() << "\n";
    else
        cout << ac1.owner()
            << " is poorer than "
            << ac2.owner() << "\n";
    return 0;
```

```
}
```

Harjoitus 13.8

Olkoon seuraava luokka Y, jossa on kaksi staattista tietojäsentä ja jäsenfunktiota:

```
class X {
public:
    X( int i ) { _val = i; }
    int val() { return _val; }
private:
    int _val;
};

class Y {
public:
    Y( int i );
    static X xval();
    static int callsXval();
private:
    static X _xval;
    static int _callsXval;
};
```

Alusta `_xval` arvolla 20 ja `_callsXval` arvolla 0.

Harjoitus 13.9

Käytä harjoituksen 13.8 luokkia ja toteuta nuo kaksi Y-luokan staattista jäsenfunktiota. Niistä `callsXval()` yksinkertaisesti pitää lukua siitä, kuinka monta kertaa `xval()`:ia on kutsuttu.

Harjoitus 13.10

Mitkä seuraavista staattisten tietojäsenten esittelyistä ja määrittelyistä ovat virheellisiä, vai onko yksikään? Selitä, miksi.

```
// example.h
class Example {
public:
    static double rate = 6.5;

    static const int vecSize = 20;
    static vector<double> vec(vecSize);
};

// example.C
#include "example.h"
double Example::rate;
vector<double> Example::vec;
```

13.6 Osoitin luokan jäseneen

Oletetaan, että Screen-luokkamme määrittelee neljä uutta jäsenfunktiota — forward(), back(), up() ja down() — jotka siirtävät vastaavasti kohdistinta näytöllä yhden position oikealle, vasemmalle, ylös tai alas. Ensiksi pitää esitellä nämä uudet jäsenfunktiot luokan rungossa:

```
class Screen {
public:
    inline Screen& forward();
    inline Screen& back();
    inline Screen& end();
    inline Screen& up();
    inline Screen& down();
    // muut jäsenfunktiot kuten ennen
private:
    inline int row();
    // muut yksityiset jäsenet kuten ennen
};
```

Jäsenfunktiot forward() ja back() siirtävät kohdistinta yhden merkin kerrallaan. Jos kohdistin saavuttaa näytön oikean alakulman tai vasemman yläkulman, se kiertyy ympäri.

```
inline Screen& Screen::forward()
{ // siirrä _cursor:ia yksi näytön elementti eteenpäin

    ++_cursor;

    // tarkista näytön loppu; kierrä ympäri
    if ( _cursor == _screen.size() )
        home();

    return *this;
}

inline Screen& Screen::back()
{ // siirrä _cursor:ia yksi näytön elementti taaksepäin

    // tarkista näytön alku; kierrä ympäri
    if ( _cursor == 0 )
        end();
    else
        --_cursor;

    return *this;
}
```

end() asettaa kohdistimen näytön oikeaan alakulmaan. Se täydentää aikaisemmin esiteltyä jäsenfunktiota home():

```
inline Screen& Screen::end()
{
    _cursor = _width * _height - 1;
    return *this;
}
```

up() ja down() siirtävät kohdistinta ylös ja alas yhden näytön rivin. Kun kohdistin saavuttaa näytön ylimmän tai alimman rivin, se ei kierry ympäri, vaan antaa sen sijaan summeriäänen ja jää paikalleen:

```
const char BELL = '\007';

inline Screen& Screen::up()
{ // siirrä _cursor:ia ylös näytön yksi rivi
  // älä kierrä ympäri; soita summeria sen sijaan

    if ( row() == 1 ) // ylin rivi?
        cout << BELL << endl;
    else
        _cursor -= _width;

    return *this;
}

inline Screen& Screen::down()
{
    if ( row() == _height ) // alin rivi?
        cout << BELL << endl;
    else
        _cursor += _width;

    return *this;
}
```

row() on yksityinen jäsenfunktio, joka tukee up()- ja down()-toteutuksia, jotka palauttavat kohdistimen nykyisen rivin position:

```
inline int Screen::row()
{ // palauta nykyinen rivi
    return ( _cursor + _width ) / _width;
}
```

Screen-luokan käyttäjät ovat pyytäneet tekemään repeat()-funktion, joka tekee joitakin käyttäjän määrittämiä operaatioita n kertaa. Epäyleispätevä toteutus voisi olla seuraavanlainen:

```
Screen &repeat( char op, int times )
{
    switch( op ) {
        case DOWN: // käynnistä Screen::down() n kertaa
            break;
        case UP:   // käynnistä Screen::up() n kertaa
            break;
        // ...
    }
}
```

Vaikka tämä toteutus toimii, siinä on useita huonoja puolia. Eräs ongelma on sen luottamus siihen, että Screen-luokan jäsenfunktiot säilyvät muuttumattomina. Joka kerta, kun jäsenfunktio lisätään tai poistetaan, pitää repeat() ylläpitää. Toinen ongelma on sen koko. Koska sen pitää testata jokainen mahdollinen jäsenfunktio, repeat():in koko listaus näyttää suurelta ja tarpeettoman monimutkaiselta.

Vaihtoehtona on yleisempi ratkaisu korvata op parametrilla, joka on osoitin Screen-jäsenfunktioon -tyyppinen. repeat():in ei enää tarvitse päätellä aiottua toimenpidettä. Koko switch-lause voidaan poistaa. Osoittimet luokan jäseniin, niiden määrittelyt sekä käyttö ovat seuraavien alikohtien aiheena.

13.6.1 Luokan jäsenen tyyppi

Osoittimeen, joka osoittaa funktioon, ei tulisi sijoittaa jäsenfunktion osoitetta, vaikka niiden molempien paluutyyppi ja parametriluettelo olisivat täsmälleen samanlaiset. Esimerkiksi seuraavassa pfi on osoitin funktioon, jolla ei ole parametreja ja jonka paluutyyppi on int:

```
int (*pfi)();
```

Tällöin, jos on kaksi globaalia funktiota, HeightIs() ja WidthIs()

```
int HeightIs();
int WidthIs();
```

jommankumman tai molempien funktioiden sijoitus pfi:hin on sallittua ja oikein:

```
pfi = HeightIs;
pfi = WidthIs;
```

Screen-luokassa on myös määritelty kaksi käsittelyfunktiota — height() ja width() — joilla ei myöskään ole parametreja ja joiden paluutyyppi on int:

```
inline int Screen::height() { return _height; }
inline int Screen::width()  { return _width; }
```

Kummankin jäsenfunktion, height() tai width(), sijoitus pfi:hin on kuitenkin tyyppivirhe, josta aiheutuu käännöksenaikainen virhe:

```
// sijoitus ei sallita: tyyppivirhe
pfi = &Screen::height;
```

Miksi tapahtuu tyyppivirhe? Jäsenfunktiolla on lisätyyppiominaisuus, joka puuttuu muista funktioista — *sen luokka*. Osoittimen jäsenfunktioon pitää täsmätä funktion kanssa, joka siihen sijoitetaan, ei vain kahdessa, vaan kolmessa asiassa: (1) parametrien tyyppi ja lukumäärä, (2) paluutyyppi ja (3) luokkatyyppi, jonka jäsen se on.

Tyyppivirhe osoittimen jäsenfunktioon ja osoittimen funktioon välillä johtuu näiden kahden osoittimen esitystavan eroista. Osoittimeen, joka osoittaa funktioon, tallennetaan funktion osoite, jota voidaan käyttää tuon funktion kutsumiseen suoraan. (Osoittimet funktioihin on käsitelty kohdassa 7.9.) Osoitin jäsenfunktioon pitää ensin sitoa olioon tai sen pitää saada this-osoitin funktion käynnistämistä varten, ennen kuin funktiota, johon se viittaa, voidaan kutsua. (Seuraavissa alikohdissa näemme, kuinka osoitin jäsenfunktioon sidotaan olioon tai osoittimella kutsutaan jäsenfunktiota.) Vaikka sekä tavallista osoitinta funktioon että osoitinta jäsenfunktioon kutsutaan funktioiksi, ne ovat kuitenkin erilaisia.

Kun osoitin jäsenfunktioon esitellään, se vaatii laajempaa syntaksia, jossa otetaan luokan tyyppi huomioon. Sama pätee myös osoittimille luokan tietojäseniin. Mietitäänpä Screen-luokan jäsentä _height. Sen täydellinen tyyppi on Screen-luokan jäsen tyyppiä short. Niin muodoin osoittimen tyyppi _height-tietojäseneen on osoitin Screen-luokan jäseneen, joka on tyyppiä short. Tämä on kirjoitettu seuraavassa:

```
short Screen::*
```

Osoittimen määrittely Screen-luokan short-tyyppiseen jäseneen näyttää tältä:

```
short Screen::*ps_Screen;
```

ps_Screen voidaan alustaa _height-tietojäsenen osoitteella kuten seuraavassa:

```
short Screen::*ps_Screen = &Screen::_height;
```

Samalla tavalla siihen voidaan sijoittaa _width:in osoite kuten tässä:

```
ps_Screen = &Screen::_width;
```

ps_Screen:iin voidaan asettaa joko _width tai _height, koska molemmat ovat Screen-luokan short-tyyppisiä tietojäseniä.

Tyyppivirhe osoittimen tietojäseneen ja tavallisen osoittimen välillä johtuu myös näiden kahden osoittimen erilaisesta esitystavasta. Tavallinen osoitin sisältää kaiken tarvittavan tiedon olioon viittausta varten. Osoitin tietojäseneen pitää ensin sitoa olioon tai osoittimeen, ennen kuin sitä voidaan käyttää tietojäsenen käsittelyyn. (Seuraavassa alikohdassa näemme, kuinka osoitin tietojäseneen sidotaan olioon tai osoittimeen.) (Tämän *C++*-kirjan oheislukemistossa, *Inside the C++ Object Model* ([LIPPMAN96a]), käsitellään myös osoitin jäseneen -esitystapoja.)

Osoitin jäsenfunktioon määritellään kirjoittamalla funktion paluutyyppi, parametriluettelo ja luokka. Esimerkiksi osoittimella Screen-luokan jäsenfunktioon, jolla pystytään viittaamaan jäsenfunktioihin height() ja width(), on seuraavanlainen tyyppi:

```
int (Screen::*)()
```

Tämä tyyppi määrittää osoittimen Screen-luokan jäsenfunktioon, jolla ei ole parametreja, ja palauttaa arvon, joka on tyyppiä int.

Osoitin jäsenfunktioon voidaan esitellä, alustaa ja siihen voidaan sijoittaa kuten seuraavassa:

```
// kaikkiin osoittimiin luokan jäseniin voidaan sijoittaa arvo 0
int (Screen::*pmf1)() = 0;
int (Screen::*pmf2)() = &Screen::height;

pmf1 = pmf2;
pmf2 = &Screen::width;
```

Typedef-nimen käyttö voi saada osoitin jäseneen -syntaksin helpommaksi lukea. Esimerkiksi seuraava typedef määrittelee, että Action on vaihtoehtoinen tyyppinimi seuraavalle tyypille

```
Screen& (Screen::*)()
```

joka tarkoittaa osoitinta Screen-luokan jäsenfunktioon, jolla ei ole parametreja ja palauttaa viittauksen Screen-luokan olioon.

```
typedef Screen& (Screen::*Action)();

Action default = &Screen::home;
Action next = &Screen::forward;
```

Tyyppiä osoitin jäsenfunktioon voidaan käyttää, kun esitellään funktion parametreja ja funktion paluutyyppejä. Parametrityypille osoitin jäsenfunktioon voidaan määrittää oletusargumentti. Esimerkiksi:

```
Screen& action( Screen&, Action );
```

action() on esitelty niin, että se saa kaksi parametria: viittauksen Screen-luokan olioon ja osoittimen Screen-luokan jäsenfunktioon, jolla ei ole parametreja ja palauttaa viittauksen Screen-luokkaolioon. action() voidaan käynnistää millä tahansa seuraavista tavoista:

```
Screen myScreen;
typedef Screen& (Screen::*Action)();
Action default = &Screen::home;

extern Screen& action( Screen&, Action = &Screen::display );

void ff()
{
    action( myScreen );
    action( myScreen, default );
    action( myScreen, &Screen::end );
}
```

Seuraavassa alikohdassa käsitellään osoittimen jäsenfunktioon käynnistys ja käyttö.

13.6.2 Osoittimen käyttö luokan jäseneen

Osoittimia luokan jäseniin pitää aina käsitellä tietyn luokkatyypin olion tai osoittimen olioon kautta. Teemme sen käyttämällä kahta osoitin jäseneen -operaattoria (operaattoria .* luokkaolioille ja viittauksille sekä operaattoria ->* osoittimille luokkaolioihin). Esimerkiksi jäsenfunktio käynnistetään osoittimen jäsenfunktioon kautta seuraavasti:

```
int (Screen::*pmfi)() = &Screen::height;
Screen& (Screen::*pmfS)( const Screen& ) = &Screen::copy;

Screen myScreen, *bufScreen;

// jäsenfunktion suora käynnistys
if ( myScreen.height() == bufScreen->height() )
    bufScreen->copy( myScreen );

// samanarvoinen käynnistys osoittimien jäseniin kautta
if ( (myScreen.*pmfi)() == (bufScreen->*pmfi)() )
    (bufScreen->*pmfS)( myScreen );
```

Kutsut

```
(myScreen.*pmfi)()
(bufScreen->*pmfi)()
```

vaativat sulut, koska kutsuoperaattorin — () — sidontajärjestys on korkeampi kuin osoitin jäseneen -operaattorien. Ilman sulkuja

```
myScreen.*pmfi()
```

voitaisiin tulkita tarkoittavan

```
myScreen.*(pmfi())
```

Se käynnistäisi funktion pmfi() ja sitoisi paluuarvonsa osoitin jäseneen -olion operaattoriin (.*). Tietystikään pfmi:n tyyppi ei tue sellaista käyttöä, ja se saisi aikaan käännöksenaikaisen virheen.

Osoittimia tietojäseniin käsitellään samalla tavalla seuraavasti:

```
typedef short Screen::*ps_Screen;
Screen myScreen, *tmpScreen = new Screen( 10, 10 );

ps_Screen pH = &Screen::_height;
ps_Screen pW = &Screen::_width;

tmpScreen->*pH = myScreen.*pH;
tmpScreen->*pW = myScreen.*pW;
```

Tässä on toteutus repeat()-jäsenfunktiosta, jota käsittelimme tämän kohdan alussa. Sitä on muokattu niin, että se saa osoittimen jäsenfunktioon:

```
typedef Screen& (Screen::*Action)();

Screen& Screen::repeat( Action op, int times )
{
    for ( int i = 0; i < times; ++i )
        (this->*op)();
    return *this;
}
```

Parametri op on osoitin jäsenfunktioon, joka viittaa jäsenfunktioon, jota tullaan kutsumaan times kertaa.

Esittely, jossa halutaan antaa oletusargumentit repeat():in parametreille, voisi näyttää tältä:

```
class Screen {
public:
    Screen &repeat( Action = &Screen::forward, int = 1 );
    // ...
};
```

repeat():in käynnistykset voisivat näyttää tältä:

```
Screen myScreen;
myScreen.repeat(); // repeat( &Screen::forward, 1 );
myScreen.repeat( &Screen::down, 20 );
```

Myös osoitintaulukko jäsenfunktioihin voidaan määritellä. Seuraavassa esimerkissä Menu on osoitintaulukko Screen-luokan jäsenfunktioihin, jotka suorittavat kohdistimen liikuttelua. CursorMovements on lueteltu joukko, jossa on viittaukset Menu:un:

```
Action Menu[] = {
    &Screen::home,
    &Screen::forward,
    &Screen::back,
    &Screen::up,
    &Screen::down,
    &Screen::end
};

enum CursorMovements {
    HOME, FORWARD, BACK, UP, DOWN, END
};
```

Voimme määritellä move():sta ylikuormitetun ilmentymän, joka saa CursorMovements-parametrin, joka käyttää Menu-taulukkoa kutsuakseen valittua jäsenfunktiota. Tässä on sen toteutus:

```
Screen& Screen::move( CursorMovements cm )
{
    ( this->*Menu[ cm ] )();
    return *this;
}
```

Indeksioperaattorilla ([]) on korkeampi sidontajärjestys kuin osoitin jäseneen -operaattorilla (->*). Ensimmäinen lause move():ssa valitsee ensiksi kutsuttavan jäsenfunktion indeksoimalla Menu-taulukkoa. Sitten se kutsuu jäsenfunktiota käyttämällä this-osoitinta ja osoitin jäseneen -operaattoria. Jäsenfunktiota move() voitaisiin käyttää vuorovaikutteisessa ohjelmassa, jossa käyttäjä valitsee kohdistimen liikkeen näytöllä olevasta valikosta.

13.6.3 Osoittimet staattisiin luokan jäseniin

On olemassa ero osoittimien ei-staattisiin luokan jäseniin ja osoittimien staattisiin luokan jäseniin välillä. Osoitin luokan jäseneen -syntaksia ei käytetä, kun viitataan luokan staattiseen jäseneen. Staattiset luokan jäsenet ovat globaaleja olioita ja funktioita, jotka kuuluvat luokalle. Osoittimet näihin ovat tavallisia osoittimia. (Muista, että staattisella jäsenfunktiolla ei ole this-osoitinta.)

Kun esitellään osoitin staattiseen luokan jäseneen, se näyttää samalta kuin osoitin luokattomaan jäseneen. Osoittimen käyttö käänteisesti ei vaadi luokkaoliota. Katsokaamme esimerkiksi Account-luokkaa jälleen:

```
class Account {
public:
    static void raiseInterest( double incr );
    static double interest() { return _interestRate; }
    double amount() { return _amount; }
private:
    static double _interestRate;
    double _amount;
    string _owner;
};

inline void Account::raiseInterest( double incr )
{
    _interestRate += incr;
}
```

&_interestRate:n tyyppi on double*; se ei ole

```
// virheellinen tyyppi &_interestRate:lle
double Account::*
```

Osoitin _interestRate:en määritellään seuraavasti:

```
// OK:  double*  not  double Account::*
double *pd = &Account::_interestRate;
```

Sitä on käytetty käänteisesti samaan tapaan kuin tavallista osoitinta. Se ei vaadi kyseistä luokkaoliota. Esimerkiksi:

```
Account unit;
// käyttää tavallista käänteisoperaattoria
double daily = *pd /365 * unit._amount;
```

Koska kuitenkin sekä _interestRate että _amount ovat yksityisiä jäseniä, pitää käyttää sen sijaan staattista jäsenfunktiota interest() ja ei-staattista jäsenfunktiota amount().

Osoittimen tyyppi interest()-jäsenfunktioon on sama kuin tavallinen osoitin funktioon

```
// oikein
double (*)()
```

eikä osoitin Account-luokan jäsenfunktioon:

```
// virheellinen
double (Account::*)()
```

Osoittimen määrittely ja epäsuora kutsu interest():iin käsitellään myös samalla tavalla kuin luokattomilla osoittimilla:

```
// ok: double(*pf)()  eikä   double(Account::*pf)()
double (*pf)() = &Account::interest;

double daily = pf() / 365 * unit.amount();
```

Harjoitus 13.11

Mikä on Screen-luokan _screen- ja _cursor-jäsenten tyyppi?

Harjoitus 13.12

Määrittele ja alusta osoitin jäseneen arvolla Screen::_screen. Määrittele ja sijoita osoitin jäseneen arvolla Screen::_cursor.

Harjoitus 13.13

Määrittele typedef-nimi jokaiselle Screen-luokan eri jäsenfunktiolle.

Harjoitus 13.14

Osoittimet jäseniin voidaan esitellä myös luokan tietojäseninä. Muokkaa Screen-luokan määrittelyä niin, että se sisältää osoittimen Screen-luokan jäsenfunktioon, joka on samaa tyyppiä kuin home() ja end().

Harjoitus 13.15

Muokkaa olemassa olevaa Screen-luokan muodostajaa (tai esittele uusi muodostaja), joka saa parametrin tyyppiä osoitin Screen-luokan jäsenfunktioon, jonka parametriluettelo ja paluutyyppi ovat samat kuin jäsenfunktioilla home() ja end(). Anna oletusargumentti tälle parametrille. Käytä tätä parametria alustaaksesi tietojäsenen, joka esiteltiin harjoituksessa 3.14. Salli, että Screen-luokan jäsenfunktio antaa käyttäjän asettaa tämä jäsenen.

Harjoitus 13.16

Määrittele repeat():ista ylikuormitettu ilmentymä, joka saa cursorMovements-tyyppisen parametrin.

13.7 Yhdiste: tilaa säästävä luokka

Yhdiste (*union*) on erikoislaatuinen luokka. Yhdisteessä tietojäsenet on tallennettu muistiin niin, että ne menevät toistensa kanssa päällekkäin. Jokainen jäsen alkaa samasta muistiosoitteesta. Yhdisteelle varattu muistimäärä on se, joka tarvitaan sen suurimmalle tietojäsenelle. Arvo voi olla sijoitettuna kerrallaan vain yhteen jäseneen.

Katsokaamme esimerkkiä, joka kuvastaa, miksi ja miten yhdistettä käytetään. Kääntäjän

sanojen analysointiosa erottelee käyttäjän ohjelman sanasiksi (tokens). Lause

```
int i = 0;
```

konvertoidaan viiden sanasen sarjaksi:

1. Tyypin avainsana int
2. Tunnus i
3. Operaattori =
4. Vakio 0, joka on int-tyyppinen
5. Puolipiste ;

Nämä sanaset välitetään sanojen analysoinnista jäsentäjälle. Jäsentäjän ensimmäinen vaihe on yksilöidä saamansa sanasten sarja. Annetun informaation pitää mahdollistaa, että jäsentäjä voi tunnistaa sanasten sarjan esittelyksi. Tästä syystä jokaisella sanasella on siihen liittyvää tietoa, joka mahdollistaa jäsentäjän tunnistaa edellisen sanasten sarjan seuraavasti:

```
Type ID Assign Constant Semicolon
```

Kun jäsentäjä yksilöi sanasten sarjan esittelyksi, se analysoi sitten jokaisen sanasen arvon. Tässä tapauksessa se päättelee, että

```
Type <==> int
ID <==> i
Constant <==> 0
```

Se ei tarvitse enempää tietoa Assign- ja Semicolon-osista, koska näillä sanasilla on vain kaksi mahdollista arvoa: = ja ; .

Tästä syystä eräässä sanasen esitystavassa saatetaan käyttää kahta jäsentä — token ja value. token on yksilöllinen koodi, josta tunnistetaan jokin seuraavista: Type, ID, Assign, Constant tai Semicolon. Yksilöllinen koodi voi olla esimerkiksi kokonaislukuarvo, joka edustaa ID:tä arvolla 85 ja Semicolon:ia arvolla 72. value sisältää sanasen varsinaisen arvon. Esimerkiksi edellisen esittelyn ID-sanasella value sisältäisi merkkijonon i; Type-sanasella value sisältäisi int-tyypin esitystavan.

value-tietojäsenen esitystapa on ongelmallinen. Vaikka se sisältää vain yhden arvon kaikilla sanasilla, se voi sisältää useita eri tietotyyppejä. ID-sanasen yhteydessä value viittaa merkkijonoon; Constant-sanasen yhteydessä se viittaa kokonaislukuarvoon.

Eräs mahdollinen tapa esittää useita tietotyyppejä on tietysti luokan käyttö. Kääntäjän kirjoittaja voi esitellä value:n luokkatyypiksi, joka sisältää jäsenen jokaiselle mahdolliselle tietotyypille, jota value voi edustaa.

Luokan käyttö value:n esitystapana ratkaisee ongelman. Tässä kuitenkin sanasen value-arvo on vain yksi monista mahdollisista tietotyypeistä ja vain yhtä monista luokan jäsenistä käytetään. Luokkatyyppi pitää kuitenkin kumulatiivista varastoa kaikille tietojäsenille, jotka edustavat kaikkia mahdollisia tietotyyppejä. Parempi olisi, jos luokka ylläpitäisi tarpeen mukaan vain yhtä mahdollista tietotyyppiä kerrallaan eikä tallentaisi niitä kaikkia. Yhdiste mahdollistaa juuri tuon. Tässä on yhdisteen määrittely, joka edustaa sanasen tietotyyppiä:

```
union TokenValue {
    char _cval;
    int _ival;
    char *_sval;
    double _dval;
};
```

Jos TokenValue-yhdisteen suurin tietotyyppi on dval, on TokenValue-yhdisteen koko double-tyyppisen olion koko. Yhdisteen jäsenet ovat oletusarvoltaan julkisia. Yhdisteen nimeä voidaan käyttää ohjelmassa siellä, missä luokan nimeä voidaan käyttää. Esimerkiksi:

```
// olio, jonka tyyppi on TokenValue
TokenValue last_token;

// osoitin olioon, jonka tyyppi on TokenValue
TokenValue *pt = new TokenValue;
```

Yhdisteen jäseniä käsitellään luokan jäsenen käsittelyoperaattoreilla (. ja ->) aivan kuten luokan jäseniä käyttämällä yhdisteen oliota tai osoitinta yhdisteeseen ennen operaattoria. Esimerkiksi:

```
last_token._ival = 97;
char ch = pt->_cval;
```

Yhdisteen jäsenet voidaan esitellä julkisiksi, suojatuiksi tai yksityisiksi:

```
union TokenValue {
public:
    char _cval;
    // ...
private:
    int priv;
};

int main() {
    TokenValue tp;
    tp._cval = '\n'; // ok

    // virhe: main() ei voi käsitellä yksityistä jäsentä
    //        TokenValue::priv
    tp.priv = 1024;
}
```

Yhdisteessä ei voi olla staattista jäsentä tai jäsentä, joka on viittaus. Yhdisteessä ei voi olla luokkatyypin jäsentä, jossa määritellään joko muodostaja, tuhoaja tai kopioinnin sijoitusoperaattori. Esimerkiksi:

```
union illegal_members {
    Screen s;      // virhe: muodostaja on
    Screen *ps;    // ok
    static int is; // virhe: staattinen jäsen
    int &rfi;      // virhe: viittausjäsen
};
```

Yhdisteeseen voidaan määritellä jäsenfunktioita mukaan lukien muodostajat ja tuhoajat.

```
union TokenValue {
public:
    TokenValue(int ix) : _ival(ix) { }
    TokenValue(char ch) : _cval(ch) { }
    // ...
    int ival() { return _ival; }
    char cval() { return _cval; }
private:
    int _ival;
    char _cval;
    // ...
};

int main() {
    TokenValue tp(10);
    int ix = tp.ival();
    // ...
}
```

Tässä on esimerkki, kuinka TokenValue-yhdistettä voitaisiin käyttää:

```
enum TokenKind { ID, Constant /* and other tokens */ };

class Token {
public:
    TokenKind tok;
    TokenValue val;
};
```

Token-tyyppistä oliota voitaisiin käyttää seuraavasti:

```
int lex() {
    Token curToken;
    char *curString;
    int curIval;

    // ...
    case ID: // tunnus
        curToken.tok = ID;
        curToken.val._sval = curString;
        break;

    case Constant: // kokonaislukuvakio
        curToken.tok = Constant;
        curToken.val._ival = curIval;
        break;

    // ... jne.
}
```

Yhdisteen käytössä on se vaara, että haetaan vahingossa yhdisteestä nykyinen arvo sopimattoman tietojäsenen kautta. Jos esimerkiksi viimeinen sijoitus on kohdistunut _ival:iin, ei ohjelmoija halua hakea tuota arvoa _sval-jäsenen kautta. Jos niin tehdään, se johtaa varmasti ohjelmavirheeseen.

Jotta tämänlaatuiseen virheeseen voitaisiin varautua, tulisi ohjelmoijan määritellä lisäolio, jonka tarkoitus on pitää kirjaa tyypistä, joka kulloinkin on tallennettuna yhdisteeseen. Tätä lisäoliota kutsutaan yhdisteen *erottelijaksi* (*discriminant*). Tämä on tok-jäsenen tehtävä Token-luokassa. Esimerkiksi:

```
char *idVal;
// varmista erottelijan arvo ennen sval:iin viittausta
if ( curToken.tok == ID )
    idVal = curToken.val._sval;
```

Hyvä käytäntö yhdisteen olion käsittelyssä on, että jokaiselle yhdisteen tietotyypille on joukko käsittelyfunktioiota. Esimerkiksi:

```
#include <cassert>
// käsittelyfunktio yhdisteen sval-jäsenelle
string Token::sval() {
    assert( tok==ID );
    return val._sval;
}
```

Kun yhdistettä määritellään, sen nimi on valinnainen. Jos yhdisteen nimeä ei käytetä tyyppinimenä ohjelmassa muiden olioiden esittelemiseen, ei nimeä ole syytä antaa, kun yhdisteen tyyppi määritellään. Esimerkiksi seuraava Token-määrittely on samanarvoinen edellisen määrittelynsä kanssa. Ainoa ero on, että yhdiste on ilman nimeä:

```
class Token {
public:
    TokenKind tok;
    // yhdisteen nimi jätetty pois
    union {
        char _cval;
        int _ival;
        char *_sval;
        double _dval;
    } val;
};
```

Tätä erityistä yhdisteen ilmentymää sanotaan *nimettömäksi yhdisteeksi* (*anonymous union*). Nimetön yhdiste on sellainen, jonka nimi *ei* tule olion määrittelyn jälkeen. Esimerkiksi tässä on Token-luokan määrittely, joka sisältää nimettömän yhdisteen:

```
class Token {
public:
    TokenKind tok;
    // nimetön yhdiste
    union {
        char _cval;
        int  _ival;
        char *_sval;
        double _dval;
    };
};
```

Nimettömän yhdisteen tietojäseniä voidaan käsitellä suoraan sillä viittausalueella, jossa nimetön yhdiste on määritelty. Tässä on esimerkiksi lex()-funktio uudelleenkoodattuna ja käyttää Token-luokan määrittelyä, joka puolestaan sisältää nimettömän yhdisteen:

```
int lex() {
    Token curToken;
    char *curString;
    int curIval;

    // ... selvitä, mikä sananen on
    // ... aseta nyt curToken
    case ID:
        curToken.tok = ID;
        curToken._sval = curString;
        break;
```

```
        case Constant: // kokonaislukuvakio
            curToken.tok = Constant;
            curToken._ival = curIval;
            break;
        // ... jne.
}
```

Nimetön yhdiste poistaa yhden tason jäsenen käsittelyoperaattoreista, koska yhdisteen jäsenten nimiä käsitellään aivan kuin Token-luokan jäseninä. Nimettömällä yhdisteellä ei voi olla yksityisiä tai suojattuja jäseniä eikä siihen voi määritellä jäsenfunktioita. Nimetön yhdiste, joka on määritelty globaalille viittausalueelle, pitää esitellä nimettömässä nimiavaruudessa (tai esitellä static-tyyppiseksi).

13.8 Bittikenttä: tilaa säästävä jäsen

Erityinen luokan tietojäsen, jota kutsutaan *bittikentäksi*, voidaan esitellä niin, että se voi sisältää lukuisia bittejä. Bittikentän pitää olla kokonaistyyppinen tieto. Se voi olla joko etumerkillinen tai etumerkitön. Esimerkiksi:

```
class File {
    // ...
    unsigned int modified : 1; // bittikenttä
};
```

Bittikentän tunnuksen jälkeen tulee kaksoispiste ja sen jälkeen vakiolauseke, joka määrittää bittien lukumäärän. Esimerkiksi modified on bittikenttä, joka muodostuu yhdestä bitistä.

Bittikentät, jotka on määritelty peräkkäisessä järjestyksessä luokan rungossa, on mahdollisuuksien mukaan pakattu saman kokonaisluvun vierekkäisiin bitteihin siten, että ne muodostavat tiiviin muistitilan. Esimerkiksi seuraavassa esittelyssä on tallennettu yhteen unsigned int -kokonaislukuun viisi bittikenttää, joista ensimmäinen on bittikenttä mode.

```
typedef unsigned int Bit;

class File {
public:
    Bit mode: 2;
    Bit modified: 1;
    Bit prot_owner: 3;
    Bit prot_group: 3;
    Bit prot_world: 3;
    // ...
};
```

Bittikenttää käsitellään samalla tavalla kuin muita luokan tietojäseniä. Esimerkiksi bittikenttää, joka on luokkansa yksityinen jäsen, voidaan käsitellä vain sen oman luokan jäsenfunktioiden määrittelyistä ja luokan ystävistä:

```
void File::write()
{
    modified = 1;
    // ...
}

void File::close()
{
    if ( modified )
        // ... tallenna sisältö
}
```

Tässä on yksinkertainen esimerkki, kuinka bittikenttää, joka on suurempi kuin 1 bitti, voidaan käyttää (kohdassa 4.11 käsitellään biteittäisiä operaattoreita, joita tässä esimerkissä on käytetty):

```
enum { READ = 01, WRITE = 02 }; // Tiedostotilat

int main() {
    File myFile;

    myFile.mode |= READ;
    if ( myFile.mode & READ )
        cout << "myFile.mode is set to READ\n";
}
```

Tyypillistä on, että määritellään joukko välittömiä jäsenfunktioita, joilla testataan bittikenttäjäsenen arvoa. Esimerkiksi File-luokkaan voitaisiin määritellä jäsenet isRead() ja isWrite().

```
inline int File::isRead() { return mode & READ; }
inline int File::isWrite() { return mode & WRITE; }

if ( myFile.isRead() ) /* ... */
```

Näillä jäsenfunktioilla voidaan bittikentät määritellä nyt File-luokan yksityisiksi jäseniksi.

Osoiteoperaattoria (&) ei voi käyttää bittikenttään, jolloin ei voi olla osoittimiakaan, joilla viitattaisiin luokan bittikenttiin. Eikä bittikenttä voi olla luokkansa staattinen jäsen.

C++-vakiokirjastossa on bittijoukkoluokkamalli, jolla voidaan käsitellä bittijoukkoja. Sitä tulisi käyttää bittikenttien sijasta aina, kun se on mahdollista. Bittijoukkoluokkamalli ja sen operaatiot on esitelty kohdassa 4.12.

Harjoitus 13.17

Kirjoita tämän kohdan esimerkit niin, että File-luokka käyttääkin bittijoukkoluokkaa ja sen operaattoreita, jotka kuvattiin kohdassa 4.12 sen sijaan, että se esittelisi ja käsittelisi bittikenttä-

tietojäseniä suoraan.

13.9 Luokan viittausalue ◆

Luokan runko määrittelee viittausalueen. Luokan jäsenten nimet esitellään luokan rungossa luokkansa viittausalueelle.

Jäsenen käsittelyoperaattoreita (piste ja nuoli) ja viittausalueen erotteluoperaattoria (::) voidaan käyttää ohjelmissa luokan viittausalueella esiteltyjen jäsenten käsittelyyn. Kun piste- tai nuolioperaattoria käytetään, nimi ennen operaattoria ilmaisee luokkatyypin olion tai osoittimen olioon, ja nimeä, joka tulee operaattorin jälkeen, etsitään luokan viittausalueelta. Samalla tavalla, kun käytetään viittausalueen erotteluoperaattoria, jäsenen nimeä, joka tulee operaattorin jälkeen, etsitään sen luokan viittausalueelta, jonka nimi on operaattoria ennen. (Luvuissa 17 ja 18 näemme myös, kuinka johdettu luokka voi viitata kantaluokkiensa jäseniin.)

Aina ei ole välttämätöntä viitata luokan jäseniin käyttämällä jäsenen käsittelyoperaattoria tai viittausalueen erotteluoperaattoria. Tietyt ohjelman tekstiosat ovat luokan viittausalueella, jolloin luokan jäseniä voidaan käsitellä suoraan näistä ohjelman tekstiosista. Ensimmäinen osa ohjelmatekstistä, joka on luokan viittausalueella, on itse luokan määrittely. Luokan jäsenen nimeä voidaan käyttää luokkansa rungossa esittelynsä jälkeen. Esimerkiksi:

```
class String {
public:
    typedef int index_type;

    // parametrityyppi viittaa: String::index_type
    char& operator[]( index_type );
};
```

Järjestys, jossa luokan jäsenet esitellään luokan rungossa, on tärkeä. Jäseniä, jotka esitellään myöhemmin luokan rungossa, ei voi käyttää aikaisemmin esiteltyjen jäsenten esittelyissä. Jos esimerkiksi operator[]()-jäsenen esittely esiintyy ennen index_type:n typedef-nimen esittelyä, on operator[]()-esittely virheellinen, koska se käyttää esittelemätöntä index_type-nimeä:

```
class String {
public:
    // virhe: nimeä index_type ei ole esitelty
    char& operator[]( index_type );

    typedef int index_type;
};
```

On olemassa kaksi poikkeusta sääntöön, että luokan määrittelyissä käytetyt nimet pitää esitellä ennen niiden käyttöä. Ensimmäinen poikkeus koskee nimiä, joita käytetään välittömien jäsenfunktioiden määrittelyissä, toinen poikkeus koskee nimiä, joita käytetään oletusargumentteina. Tutkikaamme jokaista tilannetta vuorollaan.

Nimiresoluutio, jota käytetään välittömän jäsenfunktion määrittelyssä, tapahtuu kahdessa

vaiheessa. Ensiksi käsitellään funktion esittely (tarkoittaa funktion paluutyyppiä ja parametriluetteloa) paikassa, jossa se esiintyy luokan määrittelyssä. Sitten käsitellään funktion runko koko luokan viittausalueella — kun kaikki luokan jäsenten esittelyt on nähty. Katsokaamme nyt esimerkkiämme, jossa määritellään operator[]()-jäsen välittömäksi luokan rungossa:

```
class String {
public:
    typedef int index_type;
    char& operator[]( index_type elem )
        { return _string[ elem ]; }
private:
    char *_string;
};
```

Nimiresoluution ensimmäinen vaihe etsii operator[]()-jäsenen esittelyssä käytettyjä nimiä. Parametrin tyyppinimeä index_type etsitään tämän ensimmäisen vaiheen aikana. Koska tämä ensimmäinen vaihe tapahtuu, kun jäsenfunktion määrittely kohdataan luokan rungossa, pitää index_type-nimen olla esiteltynä ennen operator[]()-jäsenen määrittelyä.

Huomaa, että _string-jäsen on esitelty luokan rungossa operator[]()-jäsenen määrittelyn jälkeen. Tämä on ok, eikä _string ole esittelemätön nimi operator[]()-jäsenen rungossa. Nimiresoluution toisen vaiheen aikana etsitään nimiä jäsenfunktioiden rungoista välittömien jäsenfunktioiden määrittelyistä. Tämä nimiresoluutio tapahtuu luokan koko viittausalueella. On kuin jäsenfunktioiden rungot käsiteltäisiin viimeisenä juuri ennen luokan rungon loppua. Tässä vaiheessa kaikki luokan jäsenfunktiot on esitelty.

Oletusargumentit ratkaistaan myös nimiresoluution toisessa vaiheessa — luokan koko viittausalueella. Esimerkiksi clear()-jäsenfunktion esittelyssä käytetään staattisen bkground-jäsenen nimeä, joka on määritelty myöhemmin luokassa:

```
class Screen {
public:
    // bkground viittaa staattiseen jäseneen
    // esitelty myöhemmin luokan määrittelyssä
    Screen& clear( char = bkground );
private:
    static const char bkground = '#';
};
```

Vaikka jäsenfunktioiden esittelyjen oletusargumentit ratkaistaan luokan koko viittausalueella, ohjelma on virheellinen, jos oletusargumentti viittaa ei-staattiseen jäseneen. Ei-staattinen jäsen pitää sitoa luokkatyyppinsä olioon tai osoittimeen olioon ennen kuin sen arvoa voidaan käyttää ohjelmassa. Ei-staattisen tietojäsenen käyttö oletusargumenttina sotii tätä rajoitusta vastaan. Jos esimerkiksi edellinen esimerkki on kirjoitettu uudelleen näin

```
class Screen {
public:
    // ...
    // virhe: bkground ei ole staattinen jäsen
```

```
    Screen& clear( char = bkground );
private:
    const char bkground;
};
```

ratkaistaan oletusargumentin nimi ei-staattiseksi tietojäseneksi bkground, jolloin oletusargumentti on virheellinen.

Luokan jäsenten määrittelyt, jotka esiintyvät luokan rungon ulkopuolella, ovat usein ohjelmatekstin osia (luokan määrittelyn lähellä), jotka ovat luokan viittausalueella. Näissä ohjelman tekstiosissa luokan jäsenten nimet löytyvät, vaikka näitä jäseniä ei käsitellä jäsenen käsittelyoperaattorien tai viittausalueen erotteluoperaattorin kautta. Katsokaamme, kuinka nimiresoluutio etenee näissä jäsenten määrittelyissä.

Yleensä, jos luokan jäsenen määrittely esiintyy luokan rungon ulkopuolella, pidetään ohjelmatekstiä, joka tulee määritellyn jäsenen nimen jälkeen, kuuluvana luokan viittausalueeseen jäsenen määrittelyn loppuun saakka. Siirtäkäämme esimerkiksi operator[]():in määrittely String-luokan ulkopuolelle:

```
class String {
public:
    typedef int index_type;
    char& operator[]( index_type );
private:
    char *_string;
};

// operator[]() käsittelee näitä: index_type ja _string
inline char& String::operator[]( index_type elem )
{
    return _string[ elem ];
}
```

Huomaa, että parametriluettelo viittaa suoraan typedef-jäseneen index_type tarkentamatta nimeään luokalla String::. Ohjelman teksti, joka tulee String::operator[]-jäsennimen jälkeen jäsenfunktion määrittelyn loppuun saakka, on luokan viittausalueella. Tyyppien, jotka on esitelty String-luokan viittausalueella, katsotaan ratkaisevan funktion parametriluettelossa käytettyjen tyyppien nimet.

Staattisten tietojäsenten määrittelyt esiintyvät myös luokan määrittelyn ulkopuolella. Näissä määrittelyissä ohjelmatekstin, joka tulee määritellyn staattisen jäsenen nimen jälkeen, ajatellaan myös olevan luokan viittausalueella jäsenfunktion määrittelyn loppuun saakka. Esimerkiksi staattisen jäsenen alustaja voi viitata suoraan luokan jäseniin ilman tarvetta käyttää piste-, nuoli- tai viittausalueen resoluutio-operaattoreita:

```
class Account {
    // ...
private:
    static double _interestRate;
    static double initInterest();
```

```
};

// viittaa jäsneen Account::initInterest()
double Account::_interestRate = initInterest();
```

_interestRate:n alustaja käynnistää jäsenfunktion Account::initInterest(), vaikka viittaukseen initInterest():iin ei käytetäkään tarkennettua nimeä.

Ei vain alustaja, vaan kaikki staattisen _interestRate-jäsenen jälkeen puolipisteeseen päättyvän staattisen jäsenen määrittelyn loppuun saakka, ovat Account-luokan viittausalueella. Staattisen name-jäsenen määrittely voi sitten viitata luokan nameSize-jäseneen kuten seuraavassa:

```
class Account {
    // ...
private:
    static const int nameSize = 16;
    static const char name[nameSize];
};

// nameSize on jätetty tarkentamatta Account-luokalla
const char Account::name[nameSize] = "Savings Account";
```

Vaikka nameSize-jäsentä ei ole tarkennettu Account-luokkanimellä, ei name:n määrittely ole virheellinen. Staattisen name-tietojäsenen määrittely on luokkansa viittausalueella ja voi viitata Account-luokan jäseniin sen jälkeen, kun tarkennettu Account::name-nimi on nähty.

Luokan jäsenen määrittelyssä, joka esiintyy luokan rungon ulkopuolella, ohjelmateksti, joka on ennen määritellyn jäsenen nimeä, ei ole luokan viittausalueella. Jos tämän ohjelmatekstin pitää viitata luokan jäseneen, pitää käyttää viittausalueen erotteluoperaattoria. Jos esimerkiksi staattisen jäsenen tyyppi on Account-luokan typedef-jäsen Money, pitää nimi Money tarkentaa, kun staattinen jäsen on määritelty luokan rungon ulkopuolelle:

```
class Account {
    typedef double Money;
    // ...
private:
    static Money _interestRate;
    static Money initInterest();
};

// Money pitää tarkentaa luokalla Account::
Account::Money Account::_interestRate = initInterest();
```

Jokainen luokka huolehtii omasta viittausalueestaan. Kahdella eri luokalla on eri viittausalueet. Yleensä yhden luokan jäseniä ei voi käyttää suoraan toisen luokan jäsenten määrittelyissä, ellei luokka ole toisen kantaluokka. Periytymistä ja kantaluokkia käsitellään luvuissa 17 ja 18.

13.9.1 Nimiresoluutio luokan viittausalueella

Tietystikään nimien, joita käytetään luokan viittausalueella, ei tarvitse olla luokan jäsenten nimiä. Nimiresoluutio luokan viittausalueella löytää yhtä hyvin nimet, jotka on esitelty muilla viittausalueilla. Jos nimiresoluution aikana ei luokan viittausalueella olevaa nimeä ratkaista luokan jäsenen nimeksi, etsitään luokan tai jäsenen määrittelyn ympäristöstä esittelyä nimelle. Tässä alikohdassa katsomme, kuinka luokan viittausalueella olevia nimiä ratkaistaan.

Luokan määrittelyssä käytetty nimi (paitsi välittömien jäsenfunktioiden määrittelyt ja oletusargumentit) ratkaistaan seuraavasti:

1. Luokan jäsenten esittelyt, jotka esiintyvät ennen nimen käyttöä, otetaan huomioon.
2. Ellei kohdan 1 ratkaisu onnistu, otetaan huomioon esittelyt, jotka esiintyvät nimiavaruuden viittausalueella ennen luokan määrittelyä. (Muista, että globaali viittausalue on myös nimiavaruuden viittausalue.) (Nimiavaruudet on esitelty kohdassa 8.5.)

   Esimerkiksi:

```
typedef double Money;
class Account {
    // ...
private:
    static Money _interestRate;
    static Money initInterest();
     // ...
};
```

Kääntäjä etsii aluksi Money:n esittelyä Account-luokan viittausalueelta. Kääntäjä ottaa huomioon vain niiden jäsenten esittelyt, jotka on esitelty ennen Money-sanan käyttöä. Koska yhtäkään jäsenen esittelyä ei löydy, etsii kääntäjä sitten esittelyä Money-sanalle globaalilta viittausalueelta. Vain esittelyt, jotka sijaitsevat ennen Account-luokan määrittelyä, otetaan huomioon. Money:n globaali typedef-nimen esittely löytyy ja sen tyyppiä käytetään esittelyissä _interestRate ja initInterest().

Luokan jäsenfunktiossa oleva nimi ratkaistaan seuraavasti:

1. Esittelyt, jotka ovat jäsenfunktion paikallisella viittausalueella, otetaan huomioon ensiksi. (Paikalliset viittausalueet ja paikalliset esittelyt on käsitelty kohdassa 8.1.)
2. Ellei kohdan 1 ratkaisu onnistu, otetaan huomioon kaikki luokan jäsenten esittelyt.
3. Ellei kohdan 2 ratkaisu onnistu, otetaan huomioon esittelyt, joka sijaitsevat nimiavaruuden viittausalueella ennen jäsenfunktion määrittelyä.

Tutkikaamme nyt, kuinka välittömien jäsenfunktioiden rungoissa käytetyt nimet ratkaistaan:

```
int _height;

class Screen {
public:
    Screen( int _height ) {
```

```
        _height = 0; // mikä _height? Parametri
    }
private:
    short _height;
};
```

Kun etsitään esittelyä nimelle _height, jota on käytetty Screen-luokan muodostajan määrittelyssä, kääntäjä etsii sitä aluksi muodostajan paikalliselta viittausalueelta. Funktion parametri on esitelty oman funktionsa paikallisella viittausalueella. Nimi _height, jota on käytetty muodostajan määrittelyssä, viittaa tähän parametrin esittelyyn.

Ellei parametrin esittelyä olisi löytynyt, olisi kääntäjä etsinyt sitten luokan viittausalueelta ottaen huomioon kaikki Screen-luokan jäsenten esittelyt. Tämä etsintä löytäisi esittelyn _height-jäsenestä. Luokan _height-jäsenen sanotaan jäävän piiloon muodostajan parametrin esittelyn alle. Vaikka luokan jäsen jää piiloon, on silti mahdollista käyttää sitä muodostajan rungossa tarkentamalla jäsenen nimi luokkansa nimellä tai käyttämällä this-osoitinta eksplisiittisesti. Esimerkiksi:

```
int _height;

class Screen {
public:
    Screen( long _height ) {
        this->_height = 0; // viittaa jäseneen Screen::_height
        // myös kelvollinen:
        // Screen::_height = 0;
    }
private:
    short _height;
};
```

Olettaen, ettei parametrin esittelyä eikä jäsenen esittelyä olisi löytynyt, olisi kääntäjä sitten etsinyt ympäröivien nimiavaruuksien viittausalueilta. Esimerkissämme otetaan ne esittelyt huomioon, jotka ovat globaalilla viittausalueella näkyvissä ennen Screen-luokan määrittelyä. Tämä etsintä löytää globaalin _height-olion esittelyn. Globaalin _height-olion sanotaan jäävän piiloon luokan jäsenen esittelyn alle. Vaikka globaali olio jää piiloon, on silti mahdollista käyttää sitä muodostajan rungossa tarkentamalla globaali nimi globaalin viittausalueen erotteluoperaattorilla:

```
int _height;

class Screen {
public:
    Screen( long _height ) {
        ::_height = 0; // viittaa globaaliin olioon
    }
private:
    short _height;
};
```

Jos muodostaja on esitelty luokan määrittelyn ulkopuolella, nimiresoluution kolmas vaihe ei ota huomioon vain globaalin viittausalueen esittelyitä, jotka esiintyvät ennen Screen-luokan määrittelyä, vaan myös globaalin viittausalueen ne esittelyt, jotka esiintyvät ennen jäsenfunktion määrittelyä. Esimerkiksi:

```
class Screen {
public:
    // ...
    void setHeight( int );
private:
    short _height;
};

int verify(int);

void Screen::setHeight( int var ) {
    // var: viittaa parametriin
    // _height: viittaa luokan jäseneen
    // verify: viittaa globaaliin funktioon
    _height = verify( var );
}
```

Huomaa, että globaalin verify()-funktion esittely ei ole näkyvissä ennen Screen-luokan määrittelyä. Kuitenkin nimiresoluution kolmas vaihe ottaa huomioon nimiavaruuden viittausalueen esittelyt, jotka ovat näkyvissä ennen jäsenen määrittelyä, joten globaalin verify()-funktion esittely löytyy.

Nimi, jota on käytetty luokan staattisen jäsenen määrittelyssä, ratkaistaan seuraavasti:

1. Luokan kaikkien jäsenten esittelyt otetaan huomioon.
2. Ellei kohdan 1 ratkaisu onnistu, otetaan huomioon esittelyt, jotka esiintyvät nimiavaruuden viittausalueella ennen staattisen jäsenen määrittelyä.

2-vaiheen aikana kääntäjä ottaa huomioon nimiavaruuden viittausalueen esittelyt, jotka esiintyvät ennen staattisen tietojäsenen määrittelyä, eikä vain niitä, jotka esiintyvät ennen luokan määrittelyä.

Harjoitus 13.18

Nimeä ne ohjelmatekstin osat, joita pidetään luokan viittausalueena.

Harjoitus 13.19

Nimeä ne ohjelmatekstin osat, jotka ovat luokan viittausalueella ja joita pidetään luokan koko viittausalueena (eli jossa kaikki luokan rungossa esitellyt jäsenet otetaan huomioon).

Harjoitus 13.20

Mihin esittelyihin Type-nimi viittaa, kun sitä käytetään Exercise-luokan rungossa ja sen setVal()-

jäsenfunktion määrittelyssä? (Muista, että eri käyttötavat voivat viitata eri esittelyihin.) Mihin esittelyihin initVal-nimi viittaa, kun sitä käytetään setVal()-jäsenfunktion määrittelyssä?

```
typedef int Type;
Type initVal();

class Exercise {
public:
    // ...
    typedef double Type;
    Type setVal( Type );
    Type initVal();
private:
    int val;
};

Type Exercise::setVal( Type parm ) {
    val = parm + initVal();
}
```

Jäsenfunktion setVal() määrittely on virheellinen. Huomaatko, miksi? Tee tarvittavat muutokset niin, että Exercise-luokka käyttää globaalia typedef-nimeä Type ja globaalia initVal()-funktiota.

13.10 Sisäkkäiset luokat

Luokka voidaan määritellä toisen luokan sisään. Sellaista luokkaa kutsutaan *sisäkkäiseksi luokaksi*. Sisäkkäinen luokka on sitä ympäröivän luokan jäsen. Sisäkkäisen luokan määrittely voi esiintyä ympäröivän luokan julkisessa, suojatussa tai yksityisessä osassa.

Sisäkkäisen luokan nimi on näkyvissä ympäröivän luokan viittausalueella, mutta ei muiden luokkien tai nimiavaruuksien viittausalueilla. Tämä tarkoittaa, että sisäkkäisen luokan nimi ei törmää ympäröivällä viittausalueella olevan samanlaisen nimen kanssa. Esimerkiksi:

```
class Node { /* ... */ };

class Tree {
public:
    // Node on kapseloitu Tree:n viittausalueelle
    // Luokan viittausalueella Tree::Node peittyy ::Node
    class Node {...};

    // ok: ratkaistu sisäkkäiselle luokalle: Tree::Node
    Node *tree;
};

// Tree::Node ei näy globaalilla viittausalueella
// Node ratkaistaan Node:n globaalista esittelystä
Node *pnode;
```

```
class List {
public:
    // Node on kapseloitu List:in viittausalueelle
    // Luokan viittausalue List::Node peittää ::Node:n
    class Node {...};

    // ok: ratkaistaan luokalle: List::Node
    Node *list;
};
```

Sisäkkäisessä luokassa voi olla samanlaisia jäseniä kuin muissakin luokissa:

```
// Ei ihannemääritys: koskee luokan määrittelyä
class List {
public:
    class ListItem {
        friend class List;       // ystävän esittely
        ListItem( int val = 0 );  // muodostaja
        ListItem *next;          // osoitin omaan luokkaan
        int value;
    };
    // ...
private:
    ListItem *list;
    ListItem *at_end;
};
```

Yksityinen jäsen on sellainen, jota voidaan käsitellä vain luokkansa jäsenten määrittelyistä tai luokan ystävistä. Ympäröivällä luokalla ei ole pääsylupaa sisäkkäisen luokan yksityisiin jäseniin, ellei sitä ole esitelty sisäkkäisen luokan ystäväksi. Tästä syystä ListItem esittelee List-luokan ystäväksi, jotta List-luokan jäsenten määrittelyt voisivat käsitellä ListItem-luokan yksityisiä jäseniä. Eikä sisäkkäisellä luokalla ole mitään erityisoikeuksia ympäröivän luokan yksityisiin jäseniin. Jos haluaisimme myöntää ListItem-luokalle pääsyn Lista-luokan yksityisiin jäseniin, pitäisi ympäröivän List-luokan esitellä ListItem-luokka ystäväkseen. Edellisessä esimerkissä se ei ole ystävä eikä siitä syystä ListItem voi viitata yksityisiin List-luokan jäseniin.

Kun ListItem-luokka esitellään List-luokan julkiseksi jäseneksi, se merkitsee sitä, että sisäkkäistä luokkaa voidaan käyttää tyyppinä koko ohjelman alueella — ulkopuolella List-luokan jäsenten ja ystävien määrittelyjen. Esimerkiksi:

```
// ok: esittely globaalilla viittausalueella
List::ListItem *headptr;
```

Tämä on sallivampi kuin aioimme. Sisäkkäinen ListItem-luokka tukee List-luokkamme abstraktiota emmekä halua, että ListItem-tyyppiä päästään käsittelemään koko ohjelman alueelta. Parempi suunnitelma on määritellä sisäkkäinen ListItem-luokka List-luokan yksityiseksi jäseneksi:

```
// Ei ihannemääritys: koskee luokan määrittelyä
```

```
class List {
public:
    // ...
private:
    class ListItem {
        // ...
    };
    ListItem *list;
    ListItem *at_end;
};
```

Nyt vain List-luokan jäsenten määrittelyt ja ystävät voivat käsitellä ListItem-tyyppiä. Nyt ei enää tuota harmia tehdä kaikista ListItem-luokan jäsenistä julkisia. Koska luokka on List-luokan yksityinen jäsen, vain List-luokan jäsenet ja ystävät voivat käsitellä ListItem-luokan jäseniä. Tällä uudella suunnitelmalla ei tarvita enää ystäväesittelyä. Tässä on uusi List-luokkamme määrittely:

```
// Ok, nyt se on oikein!
class List {
public:
    // ...
private:
    // ListItem on nyt sisäkkäinen ja yksityinen tyyppi
    class ListItem {
    // ja sen jäsenet ovat nyt julkisia
    public:
        ListItem( int val = 0 );
        ListItem *next;
        int value;
    };
    ListItem *list;
    ListItem *at_end;
};
```

ListItem-muodostajaa ei ole määritelty välittömäksi ListItem-luokan määrittelyssä. Se pitää määritellä luokan määrittelyn ulkopuolelle. Mihin se voidaan määritellä? ListItem-luokan muodostaja ei ole List-luokan jäsen eikä sitä siksi voi määritellä List-luokan runkoon. ListItem-luokan muodostaja pitää määritellä globaalille viittausalueelle — viittausalueelle, joka sisältää ympäröivän luokan määrittelyn. Silloin, kun sisäkkäisen luokan jäsenfunktiota ei ole määritelty välittömäksi sisäkkäisen luokan rungossa, se pitää määritellä uloimman sulkevan luokan ulkopuolelle.

Tässä on mahdollinen määrittely ListItem-muodostajalle. Seuraava globaalin viittausalueen syntaksi ei kuitenkaan ole oikein:

```
class List {
public:
    // ...
private:
```

```
    class ListItem {
    public:
        ListItem( int val = 0 );
        // ...
    };
};

// virhe: ListItem ei ole viittausalueella
ListItem::ListItem( int val ) { ... }
```

Ongelma on, että ListItem-nimi ei näy globaalille viittausalueelle. ListItem:in käytön globaalilla viittausalueella pitää ilmaista, että ListItem on sisäkkäinen luokka List-luokan viittausalueella. Tämä tehdään tarkentamalla ListItem-luokan nimi sitä ympäröivän List-luokan nimellä. Tässä on oikea syntaksi:

```
// sisäkkäisen luokan nimi tarkennettuna ympäröivän luokan nimellä
List::ListItem::ListItem( int val ) {
    value = val;
    next = 0;
}
```

Huomaa, että vain sisäkkäisen luokan nimi on tarkennettu. Ensimmäinen tarkenne, `List::`, nimeää ympäröivän luokan. Se tarkentaa nimen, joka tulee heti perään — joka on sisäkkäinen ListItem-luokka. Toinen ListItem nimeää muodostajan, ei sisäkkäistä luokkaa. Seuraavan määrittelyn jäsenen nimi ei ole oikein:

```
// virhe: muodostajan nimi on ListItem eikä List::ListItem
List::ListItem::List::ListItem( int val ) {
    value = val;
    next = 0;
}
```

Jos ListItem olisi esitelty staattiseksi jäseneksi, olisi sen määrittely pitänyt myös tehdä globaalille viittausalueelle. Sellaisessa määrittelyssä voisi staattisen jäsenen nimi näyttää tältä:

```
int List::ListItem::static_mem = 1024;
```

Huomaa, että jäsenfunktioiden ja staattisten tietojäsenten ei tarvitse olla sisäkkäisen luokan julkisia jäseniä, jotta ne voitaisiin määritellä luokan määrittelyn ulkopuolelle. ListItem-luokan yksityiset jäsenet voidaan myös määritellä globaalille viittausalueelle.

Sisäkkäinen luokka voidaan myös määritellä ympäröivän luokan ulkopuolelle. Esimerkiksi ListItem-luokan määrittely olisi voitu tehdä globaalille viittausalueelle seuraavasti:

```
class List {
public:
    // ...
private:
    // tämä esittely vaaditaan
    class ListItem;
    ListItem *list;
```

```
    ListItem *at_end;
};

// sisäkkäisen luokan nimi tarkennettu ympäröivän luokan nimellä
class List::ListItem {
public:
    ListItem( int val = 0 );
    ListItem *next;
    int value;
};
```

Globaalissa määrittelyssä pitää sisäkkäisen ListItem-luokan nimi tarkentaa sitä ympäröivän List-luokan nimellä. Huomaa, että ListItem-luokan esittelyä ei voi jättää pois List-luokan rungosta. Globaalilla viittausalueella olevaa määrittelyä ei voi määrittää sisäkkäiselle luokalle, ellei tuota luokkaa esitellä ensiksi sitä ympäröivän luokan jäseneksi. Sisäkkäisen luokan ei tarvitse olla sitä ympäröivän luokan julkinen jäsen, jotta se voidaan määritellä globaalilla viittausalueella.

Siihen saakka, kunnes sisäkkäisen luokan määrittely on nähty, voidaan esitellä vain osoittimia ja viittauksia sisäkkäiseen luokkaan. List-luokan list- ja at_end-tietojäsenten esittelyt ovat kelvollisia, vaikka ListItem-luokka on määritelty globaalilla viittausalueella, koska nuo molemmat jäsenet ovat osoittimia. Jos jompikumpi näistä jäsenistä olisi ollut olio osoittimen sijasta, olisi jäsenen esittely List-luokassa aiheuttanut käännösvirheen. Esimerkiksi:

```
class List {
public:
    // ...
private:
    // tämä esittely vaaditaan
    class ListItem;
    ListItem *list;
    ListItem at_end; // virhe: tuntematon sisäkkäinen ListItem-luokka
};
```

Miksi joku haluaisi määritellä sisäkkäisen luokan luokkansa määrittelyn ulkopuolelle? Ehkäpä sisäkkäinen luokka tukee toteutuksen ympäröivän luokan yksityiskohtia, emmekä halua List-luokkamme käyttäjien tirkistelevän ListItem-luokan yksityiskohtia. Tästä syystä emme halua laittaa sisäkkäistä luokkaa otsikkotiedostoon, joka sisältää List-luokkamme rajapinnan. Sisäkkäisen ListItem-luokan määrittely voidaan sitten laittaa tekstitiedostoihin, jotka sisältävät List-luokan ja sen jäsenten toteutuksen.

Sisäkkäinen luokka voidaan ensin esitellä ja sitten myöhemmin määritellä ympäröivän luokan rungossa. Tämä mahdollistaa sen, että sisäkkäisen luokan jäsenet voivat viitata toisiinsa. Esimerkiksi:

```
class List {
public:
    // ...
private:
    // esittely: List::ListItem
    class ListItem;
    class Ref {
        // pli on tyyppiä List::ListItem*
        ListItem *pli;
    };
    // määrittely: List::ListItem
    class ListItem {
        // pref on tyyppiä List::Ref*
         Ref *pref;
    };
};
```

Ellei ListItem-luokkaa esitellä ennen Ref-luokan määrittelyä, on pli-jäsenen esittely virheellinen, koska ListItem-nimeä ei ole esitelty.

Sisäkkäisen luokan ei pidä käsitellä ympäröivän luokan ei-staattisia jäseniä suoraan, vaikka ne olisivat julkisia. Ympäröivän luokan ei-staattisten jäsenten kaikenlainen käsittely vaatii, että se tehdään osoittimen, viittauksen tai ympäröivän luokan olion kautta. Esimerkiksi:

```
class List {
public:
    int init( int );
private:
    class ListItem {
    public:
        ListItem( int val = 0 );
        void mf( const List & );
        int value;
        int memb;
    };
};


List::ListItem::ListItem( int val )
{
    // List::init() on List-luokan ei-staattinen jäsen
    // ja sitä pitää käyttää List-tyyppisen olion tai osoittimen kautta
    value = init( val ); // virhe: init:in kelvoton käyttöyritys
}
```

Kun luokan ei-staattisia jäseniä käytetään, pitää kääntäjän pystyä tunnistamaan olio, jolle ei-staattinen jäsen kuuluu. ListItem-luokan jäsenfunktiossa on this-osoitinta käytetty vain implisiittisesti ListItem-luokan jäseniin eikä ympäröivien luokkien jäseniin. Implisiittisestä this-osoittimesta tiedämme, että tietojäsen value viittaa olioon, jolle on kutsuttu muodostajaa. ListItem-luokan muodostajan this-osoitin on ListItem*-tyyppinen. Se, mitä tarvitaan init()-jäsenen käsittelyyn, on List-tyyppinen olio tai List*-tyyppinen osoitin.

Seuraava jäsenfunktio mf() viittaa init()-jäseneen viittausparametrin kautta. Siten tiedämme, että jäsenfunktiota init() kutsutaan oliolle, joka on määritetty funktion argumenttina:

```
void List::ListItem::mf( const List &il ) {
    memb = il.init(); // ok: viittaa jäseneen init() viittauksen kautta
}
```

Vaikka ympäröivän luokan ei-staattisten tietojäsenten käsittelyyn tarvitaan olio, osoitin tai viittaus, voi sisäkkäinen luokka käsitellä suoraan ympäröivän luokan staattisia jäseniä, tyyppinimiä tai lueteltuja joukkoja (olettaen, että nämä jäsenet ovat julkisia). Tyyppinimi on joko typedef-nimi, luetellun joukon nimi tai luokan nimi. Esimerkiksi:

```
class List {
public:
    typedef int (*pFunc)();
    enum ListStatus { Good, Empty, Corrupted };
    // ...
private:
    class ListItem {
    public:
        void check_status();
        ListStatus status; // ok
        pFunc action; // ok
        // ...
    };
    // ...
};
```

pFunc, ListStatus ja ListItem ovat jokainen sisäkkäisiä tyyppinimiä ympäröivän List-luokan viittausalueella. Kaikkiin kolmeen nimeen ja luetellun joukon, ListStatus, nimiin voidaan viitata ListItem-luokan viittausalueelta. Näihin jäseniin voidaan viitata ilman tarkennusta:

```
void List::ListItem::check_status()
{
    ListStatus s = status;
    switch ( s ) {
        case Empty: ...
        case Corrupted: ...
        case Good: ...
    }
}
```

ListItem-luokan ja sitä ympäröivän List-luokan viittausalueen ulkopuolella tarvitaan ympäröivän luokan staattisiin jäseniin, tyyppinimiin ja lueteltujen joukkojen viittaamiseen viittausalueen erotteluoperaattoria. Esimerkiksi:

```
List::pFunc myAction; // ok
List::ListStatus stat = List::Empty; // ok
```

Kun viittaamme lueteltuun joukkoon, me emme kirjoita

```
List::ListStatus::Empty
```

koska lueteltuja joukkoja voidaan käsitellä suoraan sillä viittausalueella, jossa lueteltu joukko on määritelty. Miksi? Luetellun joukon määrittely ei ylläpidä omaa viittausaluettaan kuten luokan määrittely tekee.

13.10.1 Nimiresoluutio sisäkkäisen luokan viittausalueella

Tutkikaamme nyt, kuinka nimiresoluutio etenee sisäkkäisen luokan määrittelyssä ja sen jäsenten määrittelyissä.

Nimi, jota käytetään sisäkkäisen luokan määrittelyssä (paitsi välittömän jäsenfunktion määrittelyssä ja oletusargumenttina) ratkaistaan seuraavasti:

1. Sisäkkäisen luokan jäsenten esittelyt, jotka esiintyvät ennen nimen käyttöä, otetaan huomioon.
2. Ellei 1-kohdan ratkaisu onnistu, otetaan huomioon ympäröivän luokan jäsenten esittelyt, jotka esiintyvät ennen nimen käyttöä.
3. Ellei 2-kohdan ratkaisu onnistu, otetaan huomioon esittelyt, jotka esiintyvät nimiavaruuden viittausalueella ennen sisäkkäisen luokan määrittelyä.

Esimerkiksi:

```
enum ListStatus { Good, Empty, Corrupted };
class List {
public:
    // ...
private:
    class ListItem {
    public:
        // Etsintäjärjestys:
        // 1) List::ListItem
        // 2) List
        // 3) globaali viittausalue
        ListStatus status; // viittaa globaaliin enum
        // ...
    };
    // ...
};
```

Kääntäjä etsii aluksi ListStatus-esittelyä ListItem-luokan viittausalueelta. Koska jäsenesittelyä ei löydy, kääntäjä jatkaa etsimistä List-luokan viittausalueelta. Koska esittelyä ei löydy List-luokastakaan, etsii kääntäjä sitten esittelyä globaalilta viittausalueelta. Vain ne esittelyt, jotka sijaitsevat ennen ListStatus-nimen käyttöä, otetaan etsinnässä huomioon näillä kolmella viittausalueella. Globaalin luetellun joukon esittely (enum ListStatus) löytyy ja sen tyyppiä käytetään status-olion esittelyssä.

Jos sisäkkäinen ListItem-luokka olisi määritelty globaalille viittausalueelle ympäröivän List-luokan rungon ulkopuolelle, olisi kaikki List-luokan jäsenet esitelty ja otettu huomioon:

```
class List {
private:
    class ListItem;
    // ...
public:
    enum ListStatus { Good, Empty, Corrupted };
    // ...
};

class List::ListItem {
public:
    // Etsintäjärjestys:
    // 1) List::ListItem
    // 2) List
    // 3) globaali viittausalue
    ListStatus status; // List::ListStatus
    // ...
};
```

ListStatus-nimen resoluutio etsii aluksi ListItem-luokan viittausalueelta. Koska jäsenen esittelyä ei löydy, etsii kääntäjä sitten esittelyä List-luokan viittausalueelta. Koska List-luokan täydellinen esittely on nähty, tämä etsintä ottaa huomioon kaikki List-luokan jäsenet. Tämä etsintä löytää sisäkkäisen luetellun joukon ListStatus-jäsenen List-luokasta, vaikka se on esitelty ListItem-luokan esittelyn jälkeen. status esitellään osoittimena List-luokan ListStatus-olioon. Ellei List-luokalla olisi ollut ListStatus-nimistä jäsentä, olisi etsintä jatkunut globaalilta viittausalueelta, joka esiintyy ennen sisäkkäisen ListItem-luokan määrittelyä.

Nimi, jota on käytetty sisäkkäisen luokan jäsenfunktion määrittelyssä, ratkaistaan seuraavasti:

1. Esittelyt, jotka ovat jäsenfunktion paikallisella viittausalueella, otetaan huomioon ensiksi.
2. Ellei 1-kohdan ratkaisu onnistu, otetaan huomioon kaikkien sisäkkäisten luokkien esittelyt.
3. Ellei 2-kohdan ratkaisu onnistu, otetaan huomioon ympäröivän luokan jäsenten kaikki esittelyt.
4. Ellei 3-kohdan ratkaisu onnistu, otetaan huomioon esittelyt, jotka esiintyvät nimiava-

ruuden viittausalueella ennen jäsenfunktion määrittelyä.

Mihin esittelyyn list viittaa check_status()-jäsenfunktion määrittelyssä seuraavassa koodikatkelmassa?

```
class List {
public:
    enum ListStatus { Good, Empty, Corrupted };
    // ...
private:
    class ListItem {
    public:
        void check_status();
        ListStatus status; // ok
        // ...
    };
    ListItem *list;
    // ...
};

int list = 0;
void List::ListItem::check_status()
{
    int value = list; // mikä list?
}
```

On erittäin todennäköistä, että ohjelmoija halusi, että list viittaa check_status()-jäsenfunktiossa globaaliin olioon:

- value ja globaali list-olio ovat molemmat int-tyyppisiä. List::list-jäsen on osoitintyyppi eikä sitä voi sijoittaa value-olioon ilman eksplisiittistä tyyppimuunnosta.
- Ei ole sallittua, että ListItem käsittelisi ympäröivän luokan yksityistä jäsentä kuten list.
- list on ei-staattinen tietojäsen ja siihen saa viitata ListItem-luokan jäsenfunktioista olion, osoittimen tai viittauksen kautta.

Kuitenkin, kun otetaan huomioon kaikki tuo, ratkaistaan check_status()-jäsenessä käytetty list-nimi List-luokan list-tietojäseneksi. Muista, että jos nimeä ei löydy sisäkkäisen ListItem-luokan viittausalueelta, sitä etsitään ympäröivän luokan viittausalueelta ennen globaalia viittausaluetta. Ympäröivän List-luokan list-jäsen jättää piiloon globaalin viittausalueen olion. Tästä aiheutuu virheilmoitus, koska list-osoittimen käyttö check_status()-jäsenessä ei ole sallittu.

Käsittelyoikeus ja tyypin yhteensopivuus tarkistetaan vain, jos nimi ratkaistaan. Jos nimen käyttö on virheellinen, ei nimiresoluutio etsi esittelyä, joka sopii nimen käyttöön paremmin. Sen sijaan siitä aiheutuu virheilmoitus.

Jotta globaalia list-oliota voitaisiin käyttää, pitää käyttää globaalin viittausalueen erotteluoperaattoria:

```
void List::ListItem:: check_status() {
    value = ::list; // ok
}
```

Jos sen sijaan check_status()-jäsenfunktio on määritelty välittömäksi ListItem-luokan runkoon, saa tämä viimeisin muokkaus aikaan kääntäjän virheilmoituksen ilmoittaen, että globaalin viittausalueen list-nimi on esittelemättä.

```
class List {
public:
    // ...
private:
    class ListItem {
    public:
        // virhe: ei esittelyä näkyvissä tälle: ::list
        void check_status() { int value = ::list; }
        // ...
    };
    ListItem *list;
    // ...
};

int list = 0;
```

Globaali list-olio on esitelty List-luokan määrittelyn jälkeen. Jos jäsenfunktio on määritelty välittömäksi luokan rungossa, otetaan huomioon vain globaalit esittelyt, jotka ovat näkyvissä ennen ympäröivän luokan määrittelyä. Jos check_status()-määrittely esiintyy List-luokan määrittelyn jälkeen, otetaan huomioon globaalit esittelyt, jotka ovat näkyvissä ennen check_status()-määrittelyä, jolloin list-olion globaali esittely löytyy.

Harjoitus 13.21

Luvussa 11 on toimiva esimerkki iStack-luokan käytöstä. Muuta tuota esimerkkiä niin, että esittelet poikkeusluokat pushOnFull ja popOnEmpty iStack-luokan julkisiksi sisäkkäisiksi luokiksi. Muokkaa iStack-luokan ja sen jäsenfunktioiden määrittelyjä sekä main()-funktion määrittelyä, joka esiteltiin luvussa 11, jotta se viittaisi näihin sisäkkäisiin luokkiin.

13.11 Luokat nimiavaruuden jäseninä

Tähän saakka esitetyt luokat nimiavaruuden viittausalueella on määritelty globaalin nimiavaruuden viittausalueelle. Luokkia voi määritellä myös käyttäjän esittelemiin nimiavaruuksiin. Luokan nimi, joka on määritelty käyttäjän esittelemään nimiavaruuteen, on näkyvissä vain tuon nimiavaruuden viittausalueella eikä globaalilla viittausalueella tai muissa nimiavaruuksissa. Tämä tarkoittaa, että luokan nimi ei törmää toisten nimien kanssa, jotka on esitelty muissa nimiavaruuksissa. Esimerkiksi:

```
namespace cplusplus_primer {
    class Node { /* ... */ };
}
namespace DisneyFeatureAnimation {
    class Node { /* ... */ };
}
Node *pnode; // virhe: Node ei ole näkyvissä globaalilla viittausalueella

// OK: esitellään nodeObj tyypiksi DisneyFeatureAnimation::Node
DisneyFeatureAnimation::Node nodeObj;

// using-esittely: saa Node:n näkyväksi globaalilla viittausalueella
using cplusplus_primer::Node;
Node another; // cplusplus_primer::Node
```

Kuten edellisissä kahdessa kohdassa esiteltiin, voidaan luokan jäsen (tarkoittaa jäsenfunktiota, staattista tietojäsentä tai sisäkkäistä luokkaa) määritellä luokkansa rungon ulkopuolelle. Jos kirjaston toteuttajana sijoitamme luokkamme määrittelyt käyttäjän esittelemään nimiavaruuteen, minne sijoitamme ne luokan jäsenten määrittelyt, jotka määritellään luokkien runkojen ulkopuolelle? Nämä määrittelyt voidaan sijoittaa joko nimiavaruuteen, joka sisältää uloimman luokan määrittelyn tai johonkin sitä ympäröiviin nimiavaruuksiin. Tämä mahdollistaa, että voimme järjestää kirjastomme koodin seuraavasti:

```
// --- primer.h ---
namespace cplusplus_primer {
    class List {
    // ...
    private:
        class ListItem {
        public:
            void check_status();
            int action();
            // ...
        };
    };
}

// --- primer.C ---
#include "primer.h"
```

```
namespace cplusplus_primer {
    // ok: check_status() on määritelty samassa nimiavaruudessa kuin List
    void List::ListItem::check_status() { }
}

// ok: action() on määritelty globaalille viittausalueelle,
//     nimiavaruuteen, joka ympäröi List-luokan määrittelyä
// Jäsenen nimi tarkennetaan nimiavaruuden nimellä
int cplusplus_primer::List::ListItem::action() { }
```

Sisäkkäisen ListItem-luokan jäsenet voidaan määritellä cplusplus_primer-nimiavaruuteen, joka sisältää List-luokan määrittelyn, tai globaalille viittausalueelle, joka ympäröi cplusplus_primer-nimiavaruuden määrittelyä. Molemmissa tapauksissa pitää määrittelyssä jäsenen nimi tarkentaa asianmukaisesti ympäröivien luokkien nimillä ja ympäröivien, käyttäjän esittelemien nimiavaruuksien nimillä sen ulkopuolella, jossa se on esitelty.

Kuinka nimiresoluutio etenee jäsenfunktiossa, joka esiintyy käyttäjän esittelemässä nimiavaruudessa? Kuinka esimerkiksi nimiresoluutio etenee action():in määrittelyssä, jotta löydettäisiin esittely someVal:ille?

```
int cplusplus_primer::List::ListItem::action() {
    int local = someVal;
    // ...
}
```

Jäsenfunktion määrittelyn paikalliset viittausalueet tutkitaan ensin, minkä jälkeen etsitään ListItem-luokan koko viittausalueelta, sitten etsitään List-luokan koko viittausalueelta. Tähän saakka tämä noudattaa kohdassa 13.10 kuvattua nimiresoluutiota. Sitten tutkitaan cplusplus_primer-nimiavaruuden esittelyt. Lopuksi tutkitaan globaalin viittausalueen esittelyt. Kun tutkitaan esittelyjä cplusplus_primer-nimiavaruudesta tai globaalilta viittausalueelta, otetaan huomioon vain ne esittelyt, jotka sijaitsevat ennen action()-jäsenfunktion määrittelyä. Esimerkiksi:

```
// --- primer.h ---
namespace cplusplus_primer {
    class List {
    // ...
    private:
        class ListItem {
        public:
            int action();
            // ...
        };
    };
    const int someVal = 365;
}

// --- primer.C ---
```

```
#include "primer.h"

namespace cplusplus_primer {

    int List::ListItem::action() {
        // ok: cplusplus_primer::someVal
        int local = someVal;

        // virhe: calc() on vielä esittelemättä
        double result = calc( local );
        // ...
    }

    double calc(int) { }
    // ...
}
```

Cplusplus_primer-nimiavaruuden määrittely ei ole yhtenäinen. List-luokan ja someVal-olion määrittelyt esiintyvät nimiavaruuden määrittelyn ensimmäisessä osassa, joka on sijoitettu primer.h-otsikkotiedostoon. calc()-funktion määrittely esiintyy nimiavaruuden määrittelyssä, joka on tehty primer.C-toteutustiedostoon. calc()-funktion käyttö action()-jäsenfunktiossa on virhe, koska se on esitelty sen jälkeen, kun sitä on käytetty action()-jäsenfunktion määrittelyssä. Jos calc() on osa cplusplus_primer-nimiavaruuden rajapintaa, se tulisi esitellä siinä nimiavaruuden osassa, joka esiintyy otsikkotiedostossa, kuten seuraavassa:

```
// --- primer.h ---
namespace cplusplus_primer {
    class List {
        // ...
    };
    const int someVal = 365;
    double calc(int);
}
```

Muussa tapauksessa, jos calc()-funktiota käytetään vain action()-jäsenfunktiossa helpottamaan sen toteutusta eikä se ole osa nimiavaruuden rajapintaa, se pitää esitellä ennen action()-jäsenfunktiota niin, että siihen voidaan viitata action()-jäsenfunktion määrittelyssä.

Tämä on samankaltainen esittelyiden etsintä globaalilta viittausalueelta, jonka näimme edellisissä kohdissa: esittelyt, jotka esiintyvät ennen jäsenen määrittelyä, otetaan huomioon, kun taas esittelyt, jotka sijaitsevat jäsenen määrittelyn jälkeen, jätetään huomiotta.

Tässä on pieni muistisääntö järjestykselle, jolla viittausalueita tutkitaan, kun etsitään nimeä, joka esiintyy jäsenfunktiossa, joka sijaitsee luokkansa määrittelyn ulkopuolella. Nimet, joissa jäsenen nimi on tarkennettu, ilmaisevat järjestyksen, jolla viittausalueilta etsitään. Esimerkiksi action()-jäsenen nimi edellisessä esimerkissä on tarkennettu seuraavasti:

```
cplusplus_primer::List::ListItem::action()
```

Tarkenne cplusplus_primer::List::ListItem:: ilmaisee käänteisen järjestyksen, jolla luokan viittausalueilta ja nimiavaruuden viittausalueelta etsitään. Ensimmäinen etsittävä luokan viittausalue on ListItem-luokan. Sen jälkeen etsitään sitä ympäröivän List-luokan viittausalueelta. Cplusplus_primer-nimiavaruuden viittausalueelta etsitään viimeiseksi ennen viittausaluetta, joka sisältää action()-jäsenfunktion määrittelyn. Sellaisen etsinnän aikana kaikkien luokkien viittausalueilta otetaan huomioon kaikki luokan jäsenten esittelyt, kun taas etsinnässä nimiavaruuden viittausalueelta otetaan huomioon vain esittelyt, jotka on nähty ennen jäsenen määrittelyä.

Luokka, joka on määritelty nimiavaruuden viittausalueelle, on mahdollisesti näkyvissä koko ohjelman alueella. Jos primer.h-otsikkotiedosto otetaan mukaan useampaan kuin yhteen ohjelmatekstitiedostoon, nimen cplusplus_primer::List käyttö näissä eri tiedostoissa viittaa samaan luokkaan. Luokka on ohjelman ilmentymä, jolle voi olla useampi kuin yksi määrittely ohjelmassa. Luokan määrittely pitää laittaa kerran jokaiseen tekstitiedostoon, jossa luokka tai sen jäseniä on määritelty tai käytetty. Luokan määrittelyn pitää kuitenkin olla täsmälleen samanlainen jokaisessa tekstitiedostossa, joissa se esiintyy. Tästä syystä tulisi nimiavaruuden luokkamäärittely laittaa otsikkotiedostoon kuten primer.h. Tämä otsikkotiedosto voidaan sitten ottaa mukaan jokaiseen tekstitiedostoon, jossa luokan jäseniä määritellään tai käytetään. Tämä estää eroavaisuuksien syntymisen, jos luokan määrittely on koodattu useammin kuin kerran.

Nimiavaruusluokan muut kuin välittömät funktiot ja nimiavaruusluokan staattiset tietojäsenet ovat myös ohjelman ilmentymiä. Kuitenkin näille jäsenille saa laittaa vain yhden määrittelyn koko ohjelmaan. Tästä syystä näiden jäsenten määrittelyjä ei sijoiteta otsikkotiedostoon, joka sisältää luokan määrittelyn, vaan omaan erilliseen tiedostoon kuten ohjelmatekstitiedostoon primer.C.

Harjoitus 13.22

Käytä harjoituksessa 13.21 määriteltyä iStack-luokkaa ja esittele nyt poikkeusluokat pushOnFull ja popOnEmpty LibException-nimiavaruuden jäseninä kuten seuraavassa:

```
namespace LibException {
    class pushOnFull{ };
    class popOnEmpty{ };
}
```

ja esittele iStack-luokka puolestaan Container-nimiavaruuden jäsenenä. Muokkaa iStack-luokan ja sen jäsenfunktioiden määrittelyjä sekä main()-funktion määrittelyä, jotta se viittaisi näihin luokkiin nimiavaruuden jäseninä.

13.12 Paikalliset luokat ◆A

Luokka voidaan määritellä myös funktion rungon sisään. Sellaista luokkaa kutsutaan *paikalliseksi luokaksi* (*local class*). Paikallinen luokka on näkyvissä vain paikallisella viittausalueella, jossa se on määritelty. Toisin kuin sisäkkäisellä luokalla, ei ole olemassa syntaksia, jolla viitattaisiin paikallisen luokan jäseneen sen viittausalueen ulkopuolelta, jolla paikallinen luokka on määritelty. Tästä syystä paikallisen luokan jäsenfunktiot pitää määritellä luokan määrittelyssä. Käytännössä tämä rajoittaa paikallisen luokan jäsenfunktioiden monimutkaisuutta muutamaan koodiriviin. Sen lisäksi koodista tulee lukijalleen vaikeata käsittää.

Koska ei ole olemassa syntaksia määritellä paikallisen luokan jäseniä nimiavaruuden viittausalueelle, ei paikalliseen luokkaan ole luvallista esitellä staattisia tietojäseniä.

Paikallisessa luokassa oleva sisäkkäinen luokka voidaan määritellä luokkansa ulkopuolelle. Määrittelyn pitää kuitenkin esiintyä paikallisella viittausalueella, joka sisältää ympäröivän paikallisen luokan määrittelyn. Sisäkkäisen luokan nimen paikallisen luokan viittausalueen määrittelyssä pitää tarkentaa ympäröivän luokan nimellä. Sisäkkäisen luokan esittelyä ympäröivänä luokkana ei saa jättää pois. Esimerkiksi:

```
void foo( int val )
{
    class Bar {
    public:
        int barVal;
        class nested; // sisäkkäisen luokan esittely vaaditaan
    };

    // sisäkkäisen luokan määrittely
    class Bar::nested {
        // ...
    };
}
```

Ympäröivällä funktiolla ei ole erityisoikeuksia paikallisen luokan yksityisiin jäseniin. Tätä voidaan tietysti muuttaa saattamalla ympäröivä funktio paikallisen luokan ystäväksi. Yksityiset jäsenet ovat tuskin koskaan kuitenkaan tarpeen paikallisessa luokassa. Ohjelmanosa, joka käsittelee paikallista luokkaa, on erittäin rajoitettu. Paikallinen luokka on kapseloitu paikalliselle viittausalueelleen. Lisäkapselointi tiedon piilotuksen kannalta olisi todennäköisesti liioittelua. On tuskin koskaan käytännössä syytä, miksi paikallisen luokan jäseniä ei tehtäisi julkisiksi.

Kuten sisäkkäisten luokkien yhteydessä, myös ympäröivän viittausalueen nimet, joita paikallinen luokka voi käsitellä, on rajoitettu. Paikallinen luokka voi käsitellä vain tyyppinimiä, staattisia muuttujia ja lueteltuja joukkoja ympäröiviltä paikallisilta viittausalueilta. Esimerkiksi:

```
int a, val;
```

```
void foo( int val )
{
    static int si;
    enum Loc { a = 1024, b };
    class Bar {
    public:
        Loc locVal; // ok;
        int barVal;
        void fooBar( Loc l = a ) { // ok: Loc::a
            barVal = val;    // virhe: paikallinen olio
            barVal = ::val;  // ok: globaali olio
            barVal = si;     // ok: staattinen paikallinen olio
            locVal = b;      // ok: luetellun joukon jäsen
        }
    };
    // ...
}
```

Paikallisen luokan rungon nimiresoluutio (pois lukien jäsenfunktioiden määrittelyt) ratkaistaan kirjaimellisesti etsimällä ympäröiviltä viittausalueilta esittelyitä, jotka esiintyvät ennen paikallisen luokan määrittelyä. Paikallisen luokan jäsenfunktioiden nimien ratkaisu niiden rungossa etenee niin, että ensin etsitään luokan koko viittausalueelta ennen ympäröivää viittausaluetta.

Kuten aina, jos ensimmäinen löydetty esittely osoittautuu nimen käytöltä virheelliseksi, ei enempää esittelyitä oteta huomioon. Vaikka `val`:in käyttö `fooBar()`:issa on virhe, ei globaalia `val`-muuttujaa löydy, ellei `val`-nimeä varusteta globaalin viittausalueen erotteluoperaattorilla.