15

Ylikuormitetut funktiot ja käyttäjän määrittelemät konversiot

Luvussa 15 katsotaan funktioiden kahta erityiskategoriaa: ylikuormitettuja operaattoreita ja konversiofunktioita. Näiden funktioiden avulla voidaan luokkatyyppisiä olioita käyttää lausekkeissa samalla tavalla kuin sisäisten tyyppien olioita. Tässä luvussa esitellään ensin operaattorin ylikuormituksen yleiskäsitteet ja suunnittelunäkökohdat. Sitten esitellään ystäväluokan ilmaisu, jolla on erityinen käsittelylupa ja keskustellaan, miksi ystäviä joskus tarvitaan; etenkin, kun toteutetaan ylikuormitettuja operaattoreita. Sitten luvussa katsotaan erityisiä ylikuormitettuja operaattoreita, jotka vaativat erityishuomiota, kun niitä määritellään luokkatyypille: sijoitus-, indeksi-, kutsu-, jäsenen käsittelyn nuoli-, kasvatus-, vähennys ja luokkakohtaiset new- ja delete-operaattorit. Toinen erityinen funktiokategoria on tämän luvun seuraavana kohteena: jäsenen konversiofunktiot, jotka määrittelevät “vakiokonversiot” luokkatyypille. Kääntäjä käyttää näitä konversiofunktioita implisiittisesti, kun luokkaolioita käytetään funktion argumentteina tai operandeina sisäisille tai ylikuormitetuille operaattoreille. Luku päättyy pitemmälle menevään esitykseen säännöistä, joita käytetään funktion ylikuormituksen ratkaisussa luokka-argumenttien, jäsenfunktioiden ja ylikuormitettujen operaattorien yhteydessä.

15.1 Operaattorin ylikuormitus

Kuten olemme nähneet esimerkkejä aikaisemmissa luvuissa, operaattorin ylikuormitus mahdollistaa, että ohjelmoija voi määritellä esimääritellyistä operaattoreista versioita (kuten luvussa 4 käsiteltiin) luokkatyyppisille operandeille. Esimerkiksi String-luokka, joka esiteltiin kohdassa 3.15, määrittelee monta ylikuormitettua operaattoria. Tässä on String-luokkamme määrittely:

```
#include <iostream>

class String;
istream& operator>>( istream &, String & );
ostream& operator<<( ostream &, const String & );

class String {
public:
    // ylikuormitetut muodostajat
    // tekevät automaattisen alustuksen
    String( const char * = 0 );
    String( const String & );

    // tuhoaja: automaattinen alustamisen vastakohta
    ~String();

    // ylikuormitetut sijoitusoperaattorit
    String& operator=( const String & );
    String& operator=( const char * );

    // ylikuormitettu indeksioperaattori
    char& operator[]( int ) const;

    // ylikuormitetut yhtäsuuruusoperaattorit
    // str1 == str2;
    bool operator==( const char * ) const;
    bool operator==( const String & ) const;

    // jäsenten käsittelyfunktiot
    int   size()  { return _size;   }
    char* c_str() { return _string; }
private:
    int   _size;
    char *_string;
};
```

String-luokassa on kolme ylikuormitettujen operaattorien joukkoa. Ensimmäiset määrittelevät sijoitusoperaattorit String-luokalle:

```
// ylikuormitetut sijoitusoperaattorit
String& operator=( const String & );
String& operator=( const char * );
```

Ensimmäinen sijoitusoperaattori on kopiointioperaattori, joka tukee String-tyyppisen olion sijoitusta toiseen. Kopioinnin sijoitusoperaattorit on käsitelty yksityiskohtaisesti kohdassa 14.7. Toinen sijoitusoperaattori tukee C-tyylisen merkkijonon sijoitusta String-tyyppiseen olioon kuten seuraavassa:

```
String name;
name = "Sherlock"; // use of operator=( char * )
```

Katsomme kohdassa 15.3 sijoitusoperaattoreita, jotka eivät ole kopioinnin sijoitusoperaattoreita.

Toinen ylikuormitettujen operaattoreiden joukko määrittelee yhden operaattorin — indeksi-operaattorin:

```
// ylikuormitettu indeksioperaattori
char& operator[]( int ) const;
```

Tämä operaattori mahdollistaa sen, että ohjelmat voivat indeksoida String-luokan olioita samalla tavalla kuin indeksoimme sisäisen taulukkotyypin olioita:

```
if ( name[0] != 'S' )
    cout << "oops, something went wrong\n";
```

Katsomme ylikuormitettua indeksioperaattoria tarkemmin kohdassa 15.4.

Kolmas ylikuormitettujen operaattorien joukko määrittelee String-luokan yhtäsuuruus-operaattorit. Ohjelma voi verrata kahden String-luokan olion yhtäsuuruutta tai String-luokan oliota ja C-tyylisen merkkijonon yhtäsuuruutta.

```
// yhtäsuuruusoperaattorien ylikuormitettu joukko
// str1 == str2;
bool operator==( const String & ) const;
bool operator==( const char * ) const;
```

Katsomme tätä operaattoria tarkemmin seuraavassa alikohdassa.

Ylikuormitetut operaattorit mahdollistavat sen, että luokkatyyppisiä olioita voidaan käyttää luvussa 4 määriteltyjen operaattorien kanssa ja luokkatyyppisten olioiden käsittelyyn aivan kuin ne olisivat sisäisten tyyppien olioita. Jos esimerkiksi haluamme määritellä operaation, joka tukee kahden String-tyyppisen olion yhdistämistä, voimme päättää toteuttaa tämän uuden operaation jäsenfunktiona nimeltään concat(). Mutta miksi valita nimi concat() eikä esimerkiksi append()? Vaikka valittu nimi on sekä looginen että mnemoninen, käyttäjät saattavat unohtaa täsmällisen nimen, jonka valitsimme. On usein helpompaa muistaa operaation nimi, jos määrittelemme sen ylikuormitetuksi operaattoriksi. concat()-nimen sijasta pidämme parempana nimetä uusi String-operaation näin: operator+=(). Tätä uutta operaatiota voidaan käyttää näin:

```
#include "String.h"
int main() {
    String name1 "Sherlock";
    String name2 "Holmes";

    name1 += " ";
    name1 += name2;

    if ( ! ( name1 == "Sherlock Holmes" ) )
        cout << "concatenation did not work\n";
}
```

Ylikuormitettu operaattori on esitelty luokan rungossa samalla tavalla kuin tavallinen jäsenfunktio, paitsi että sen nimi muodostuu avainsanasta operator, jonka jälkeen voi tulla jokin

suuresta C++-operaattorien esimääritellystä joukosta (katso taulukosta 15.1). Operaattori operator+=() voidaan esitellä seuraavasti String-luokassa:

```
class String {
public:
    // ylikuormitetut +=-operaattorit
    String& operator+=( const String & );
    String& operator+=( const char * );
    // ...
private:
    // ...
};
```

ja määritellä seuraavasti:

```
#include <cstring>

inline String& String::operator+=( const String &rhs )
{
    // ellei rhs:n viittaama String ole tyhjä
    if ( rhs._string )
    {
        String tmp( *this );

        // luo muistialue, joka riittää
        // sisältämään yhdistetyt String-oliot
        _size += rhs._size;
        delete[] _string;
        _string = new char[ _size + 1 ];

        // kopioi ensiksi alkuperäinen String uuteen muistipaikkaan
        // lisää sitten rhs:n viittaama String
        strcpy( _string, tmp._string );
        strcpy( _string + tmp._size, rhs._string );
    }
    return *this;
}

inline String& String::operator+=( const char *s )
{
    // ellei s ole null-osoitin
    if ( s )
    {
        String tmp( *this );

        // luo tarpeeksi suuri muistialue, joka voi
        // sisältää yhdistämisen tuloksen
        _size += strlen( s );
        delete[] _string;
        _string = new char[ _size + 1 ];
```

```
            // kopioi ensiksi alkuperäinen String uuteen muistipaikkaan
            // lisää sitten s:n viittaama C-tyylinen merkkijono
            strcpy( _string, tmp._string );
            strcpy( _string + tmp._size, s );
        }
        return *this;
}
```

15.1.1 Luokan jäsen vastaan ei-jäsen

Katsotaanpa String-luokkamme yhtäsuuruusoperaattoreita hieman tarkemmin. Ensimmäinen operaattori mahdollistaa sen, että voimme verrata kahden String-luokan olion yhtäsuuruutta, ja toinen operaattori mahdollistaa, että voimme verrata String-luokan oliota C-tyyliseen merkkijonoon. Esimerkiksi:

```
#include "String.h"

int main() {
    String flower;
    // aseta flower johonkin arvoon

    if ( flower == "lily" ) // ok
        // ...
    else
    if ( "tulip" == flower ) // virhe
        // ...
}
```

Kun yhtäsuuruusoperaattoria käytetään ensimmäisen kerran main()-funktiossa, kutsutaan String-luokan ylikuormitettua operaattoria operator==(const char *). Yhtäsuuruusoperaattorin toinen käyttökerta johtaa kuitenkin käännösvirheeseen. Kuinka se on mahdollista?

Ongelma on, että ylikuormitettu operaattori, joka on luokan jäsen, otetaan huomioon vain, kun operaattoria on käytetty *vasemman* operandin kanssa, joka on luokkatyyppinen olio. Koska vasen operandi ei ole luokkatyyppinen, kääntäjä yrittää löytää sisäisen operaattorin, joka voi ottaa vasemman operandin, joka on C-tyylinen merkkijono, ja oikean operandin, joka on String-luokka. Tietystikään sellaista operandia ei ole olemassa, ja kääntäjä antaa virheilmoituksen yhtäsuuruusoperaattorin toisesta käyttöyrityksestä main()-funktiossa.

Mutta voit sanoa, että on mahdollista luoda String-luokkaolio C-tyylisestä merkkijonosta käyttämällä luokan muodostajaa. Miksi kääntäjä ei tee seuraavaa konversiota implisiittisesti:

```
if ( String( "tulip" ) == flower ) // ok: kutsuu jäsenoperaattoria
```

Vastaus on lyhyesti: tehokkuus. Ylikuormitetut operaattorit eivät vaadi, että molemmat operandit ovat samaa tyyppiä. Esimerkiksi Text-luokassa voidaan määritellä seuraavat yhtäsuuruusoperaattorit jäsenfunktioina

```
class Text {
public:
    Text( const char * = 0 );
    Text( const Text & );

    // ylikuormitetut yhtäsuuruusoperaattorit
    bool operator==( const char * ) const;
    bool operator==( const String & ) const;
    bool operator==( const Text & ) const;

    // ....
};
```

ja lauseke voidaan kirjoittaa uudelleen main()-funktiossa seuraavasti:

```
if ( Text( "tulip" ) == flower ) // kutsu Text::operator==()
```

Joten, jotta kääntäjä löytäisi yhtäsuuruusoperaattorin tätä vertailua varten, sen pitää katsoa kaikkia luokan määrittelyjä löytääkseen kaikki muodostajat, jotka voivat konvertoida vasemman operandin luokkatyyppiseksi ja sitten löytää vastaavat ylikuormitetut yhtäsuuruusoperaattorit jokaiselle näille luokkatyypeille nähdäkseen, voiko niistä yksikään suorittaa yhtäsuuruusoperaation. Kääntäjän pitää sitten päättää, mikä muodostajan ja yhtäsuuruusoperaattorin yhdistelmä, jos yksikään, parhaiten vastaa oikeanpuoleista operandia! Jos kääntäjältä vaadittaisiin kaikki tämä, voisi C++-ohjelmien kääntämiseen kuluva aika kasvaa merkittävästi. Sen sijaan kääntäjä ottaa huomioon vain ylikuormitetut operaattorit, jotka on määritelty vasemman operandin luokassa (ja ylikuormitetut operaattorit, jotka on määritelty sen kantaluokissa, kuten tulemme näkemään luvussa 19).

On kuitenkin mahdollista esitellä ylikuormitettuja operaattoreita, jotka eivät ole luokan jäseniä. Ylikuormitettuja operaattoreita, jotka eivät ole luokan jäseniä, pidetään virheellisinä vertailuun main()-funktiossa. Tästä vertailusta, jossa C-tyylinen merkkijono on vasen operandi, voi tulla kelvollinen, jos korvaamme jäsenen yhtäsuuruusoperaattorin String-luokan yhtäsuuruusoperaattoreilla, jotka on esitelty nimiavaruuden viittausalueella, kuten seuraavassa:

```
bool operator==( const String &, const String & );
bool operator==( const String &, const char * );
```

Huomaa, että näillä globaaleilla ylikuormitetuilla operaattoreilla on yksi parametri enemmän kuin ylikuormitetuilla jäsenoperaattoreilla. Jäsenoperaattorissa käytetään implisiittistä this-osoitinta implisiittisenä ensimmäisenä parametrina. Esimerkiksi jäsenoperaattoreilla kääntäjä kirjoittaa lausekkeen

```
flower == "lily"
```

uudelleen näin

```
flower.operator==( "lily" )
```

ja vasempaan flower-operandiin viitataan ylikuormitetun jäsenoperaattorin määrityksessä this-osoittimen kautta. (this-osoitin on esitelty kohdassa 13.4.). Globaalin ylikuormitetun operaattorin parametri, joka edustaa vasenta operandia, pitää määrittää eksplisiittisesti.

Globaaleilla ylikuormitetuilla operaattoreilla String-luokan kanssa lauseke

```
flower == "lily"
```

kutsuu operaattoria:

```
bool operator==( const String &, const char * );
```

Mitä operaattoria toinen yhtäsuuruusoperaattorin käyttötilanne kutsuu?

```
"tulip" == flower
```

Emme määritelleet seuraavaa ylikuormitettua operaattoria:

```
bool operator==( const char *, const String & );
```

Pitääkö meidän? Voisimme, mutta meidän ei tarvitse. Kun ylikuormitettu operaattori on nimiavaruuden funktio, konversiot otetaan huomioon operaattorin sekä ensimmäiselle että toiselle parametrille (sekä oikealle että vasemmalle yhtäsuuruusoperaattorin operandille). Tämä merkitsee, että kääntäjä tulkitsee yhtäsuuruusoperaattorin toisen käyttökerran seuraavasti

```
operator==( String("tulip") , flower );
```

ja käynnistää seuraavan ylikuormitetun operaattorin vertailun suorittamiseen:

```
bool operator==( const String &, const String & );
```

Joten nyt ehkä ihmettelet, miksi teimme toisen ylikuormitetun operaattorin.

```
bool operator==( const String &, const char * );
```

Tyyppikonversiota C-tyylisestä merkkijonosta String-luokaksi voidaan käyttää myös oikeanpuoleiseen operandiin. main()-funktio kääntyy virheittä, jos yksinkertaisesti määrittelemme yhden nimiavaruuden ylikuormitetun operaattorin, jolle annetaan kaksi String-luokan operandia:

```
bool operator==( const String &, const String & );
```

Vastaus kysymykseen, teemmekö vain tämän yhden ylikuormitetun operaattorin vai teemmekö nämä kaksi lisäoperaattoria

```
bool operator==( const char *, const String & );
bool operator==( const String &, const char * );
```

riippuu suorituksenaikaisesta hinnasta, kun C-tyylisiä merkkijonoja konvertoidaan String-luokiksi; se siis riippuu lisäkuormasta, jonka aiheuttavat ohjelmien muodostajakutsut, jotka käyttävät String-luokkaamme. Jos ennakoimme, että yhtäsuuruusoperaattoria käytetään säännöllisesti C-tyylisten merkkijonojen ja String-tyyppisten olioiden väliseen vertailuun, voi olla hyvä ajatus tehdä kaikki kolme ilmentymää nimiavaruuden globaalista operaattorista. (Meillä on lisää sanottavaa suorituskyvystä seuraavassa kohdassa, jossa käsitellään ystäviä.)

Katsomme konversioita luokkatyypeiksi muodostajia käyttäen tarkemmin kohdassa 15.9. Tutkimme myös funktion ylikuormituksen ratkaisua uudelleen konversioiden valossa luokka-

tyypeiksi ja luokkatyypeistä kohdassa 15.10 sekä katsomme kohdassa 15.12 funktion ylikuormituksen ratkaisua, kun siihen liittyy ylikuormitettuja operaattoreita.

Sitten yleensä, kuinka voidaan päättää, tehdäänkö operaattorista luokan vai nimiavaruuden jäsen? Joissain tapauksissa ohjelmoijalla ei ole valinnanvaraa.

- Kun ylikuormitettu operaattori on luokan jäsen, jäsenoperaattori käynnistetään vain, kun operaattoria on käytetty *vasempana* operandina, joka on luokkatyyppinen. Jos operaattoria pitää käyttää vasemmalla puolella muun tyypin kanssa, silloin ylikuormitetun operaattorin *pitää* olla nimiavaruuden jäsen.
- Kieli vaatii, että sijoitus- ("="), indeksi- ("[]"), kutsu- ("()") ja jäsenen käsittelyn nuoli-operaattorit ("->") pitää määritellä luokan jäsenoperaattoreiksi. Minkä tahansa näiden operaattorien määrittely nimiavaruuden jäsenenä saa aikaan käännösvirheen. Esimerkiksi:

```
// virhe: pitää olla luokan jäsen
char& operator[]( String & ,int ix );
```

Katsomme sijoitusoperaattoria tarkemmin kohdassa 15.3, indeksioperaattoria kohdassa 15.4, kutsuoperaattoria kohdassa 15.5 ja jäsenen käsittelyoperaattoria, nuolta (->), kohdassa 15.6.

Muutoin luokan suunnittelijalla on valinta, esitelläkö operaattori luokan vai nimiavaruuden jäsenenä. Symmetriset operaattorit, kuten yhtäsuuruusoperaattori, on parasta määritellä nimiavaruuden jäseninä, jos jompikumpi operaattoreista voi olla luokkatyyppinen, kuten String-luokkamme tapauksessa.

Ennen kuin päätämme tämän alikohdan, määritelkäämme nimiavaruuden yhtäsuuruusoperaattorit String-luokalle:

```
bool operator==( const String &str1, const String &str2 )
{
    if ( str1.size() != str2.size() )
        return false;
    return strcmp( str1.c_str(), str2.c_str() ) ? false : true;
}

inline bool operator==( const String &str, const char *s )
{
    return strcmp( str.c_str(), s ) ? false : true;
}
```

15.1.2 Ylikuormitetut operaattorinimet

Vain esimääriteltyjä C++-operaattoreita voidaan ylikuormittaa. Taulukossa 15.1 on lueteltuna operaattorit, joita voidaan ylikuormittaa.

Taulukko 15.1 Ylikuormitettavat operaattorit

+	-	*	/	%	^	&	|	~
!	,	=	<	>	<=	>=	++	--
<<	>>	==	!=	&&	||	+=	-=	/=
%=	^=	&=	|=	*=	<<=	>>=	[]	()
->	->*	new	new[]	delete	delete[]			

Luokan suunnittelijan ei tulisi esitellä ylikuormitettua operaattoria, jonka nimeä ei ole tässä luettelossa. Jos esimerkiksi esitellään ylikuormitettu operaattori operator**, joka tekee eksponenttialgoritmin, aiheutuu käännösvirhe.

Seuraavaa neljää C++-operaattoria ei voida ylikuormittaa:

```
// operaattorit, joita ei voi ylikuormittaa
::   .*   .    ?:
```

Operaattorin esimääriteltyä merkitystä sisäisille tyypeille ei tulisi muuttaa. Esimerkiksi sisäistä lisäysoperaatiota ei voi korvata operaatiolla, joka tarkistaa ylivuodon.

```
// virhe: ei voida määritellä uudelleen sisäistä operaattoria int-tyypeille
int operator+( int, int );
```

Myöskään lisäoperaattoreita ei voida määritellä sisäisille tietotyypeille. Esimerkiksi operaattoria operator+, joka saa kaksi taulukkotyyppistä operandia, ei voi lisätä sisäisten operaatioiden joukkoon.

Ohjelmoija voi määritellä ylikuormitettuja operaattoreita vain operandeille, jotka ovat luokkatyyppisiä tai lueteltuja tyyppejä. Tähän pakottaa vaatimus, että ylikuormitettu operaattori voidaan esitellä joko luokan jäsenenä tai nimiavaruuden jäsenenä ja sen pitää saada vähintään yksi luokkatyyppinen tai luetellun joukon tyyppinen parametri (välitettynä joko arvona tai viittauksena).

Operaattorien esimääriteltyä sitomisjärjestystä (kohdassa 4.13 käsitellään operaattorien sidontajärjestys) ei voi korvata. Luokkatyypistä ja operaattorin toteutuksesta riippumatta

```
x == y + z;
```

suorittaa aina operaattorin operator+ ennen operaattoria operator==. Kuten esimääritellyillä operaattoreilla, kun käytetään ylikuormitettua operaattoria, sidontajärjestystä voidaan muuttaa sulkujen avulla.

Operaattorin esimääritelty *aritmeettisuus* pitää säilyttää. Unaarista loogista EI-operaattoria ei esimerkiksi voi määritellä binääriseksi operaattoriksi kahdelle String-luokkaoliolle. Seuraava toteutus ei kelpaa ja johtaakin käännösvirheeseen:

```
// kelvoton: ! on unaarinen operaattori
bool operator!( const String &s1, const String &s2 )
{
    return( strcmp( s1.c_str(), s2.c_str() ) != 0 );
}
```

Neljä esimääriteltyä operaattoria ("+", "-", "*" ja "&") toimivat sisäisille tyypeille sekä unaarisina että binäärisinä operaattoreina. Näiden operaattoreiden joko toista tai kumpaakin aritmeettisuutta voidaan ylikuormittaa.

Ylikuormitetuille operaattoreille ei oletusargumentit kelpaa paitsi operaattorille operator().

15.1.3 Ylikuormitettujen operaattoreiden suunnittelu

Sijoitus-, osoite- ja pilkkuoperaattoreilla on esimääritellyt merkitykset luokkatyyppisille operandeille. Näitä operaattoreita voidaan ylikuormittaa myös luokkaoperandeille. Kaikki muut operaattorit, joille luokan suunnittelija haluaa merkityksen luokkatyyppisen operandin käytön yhteydessä, pitää määritellä eksplisiittisesti. Valinta, mitkä operaattorit tehdään, päätellään luokan oletetuista käyttötilanteista.

Aloita aina määrittelemällä luokan julkinen rajapinta. Mitä operaatioita luokan pitää tehdä sen käyttäjille? Nämä muodostavat julkisten jäsenfunktioiden minimijoukon. Kun tämä on määritelty, on mahdollista miettiä, mitkä operaatiot tulisi määritellä ylikuormitettuina operaattoreina.

Kun luokan julkinen rajapinta on määritelty, katso loogista yhteyttä jokaisen operaation ja operaattorin välillä:

- isEmpty():stä tulee looginen EI-operaattori operator!().
- isEqual():ista tulee yhtäsuuruusoperaattori operator==().
- copy():stä tulee sijoitusoperaattori operator=().

Jokaisella operaattorilla on merkitys, joka liittyy sen esimääriteltyyn käyttöön. Esimerkiksi binäärinen + liitetään vahvasti lisäykseen. Liittämällä binäärisen +:n vastaavaan luokkatyypin operaatioon voi saada aikaan kätevän ilmaisulyhenteen. Esimerkiksi matrix-tyypin yhteenlasku, jossa lasketaan kaksi matrix-tyyppistä oliota yhteen, on tyypillinen binäärisen +-operaattorin laajennus.

Operaattorin ylikuormituksen pahin väärinkäyttö ei ole esimerkiksi määritellä vähennyslasku operaattorilla operator+(). Kukaan vastuuntuntoinen ohjelmoija ei kuitenkaan tekisi niin. Pahin operaattorin väärinkäyttö on, että se on moniselitteinen käyttäjilleen.

Tässä mielessä operaattori, jolla on moniselitteinen merkitys, on sellainen, joka tukee yhtä hyvin lukuisia eri tulkintoja. Täysin selkeä ja hyvin perusteltu selitys operator+()-operaattorista auttaa hyvin vähän String-luokkamme käyttäjiä, jotka uskovat sen toimivan yhdistelyoperaattorina. Kun ylikuormitetun operaattorin merkitys ei ole ilmeinen, on ehkä hyvä ajatus olla tekemättä sitä.

Yhtäläisyys yhdistetyn operaattorin ja vastaavien yksittäisten sisäisten tyyppien operandien operaattorien välillä (esimerkiksi yhtäläisyys seuraavien välillä: kun +-operaattorin jälkeen tulee =-operaattori ja yhdistetty +=-operaattori) pitää myös määritellä eksplisiittisesti luokalle. Jos esimerkiksi operator+()- ja operator=()-operaattorit on määritelty String-luokallemme tukemaan yhdistämistä ja jäsenittäistä kopiointia erillisinä operaatioina

```
String s1( "C" );
String s2( "++" );

s1 = s1 + s2; // s1 == "C++"
```

nämä ylikuormitetut operaattorit *eivät* tue implisiittisesti samanlaista yhdistettyä sijoitusoperaattoria:

```
s1 += s2;
```

Yhdistetty sijoitusoperaattori pitää määritellä eksplisiittisesti. Jos se on määritelty, tulisi sillä olla oletettu merkitys.

Harjoitus 15.1

Miksi seuraava *ei* käynnistä ylikuormitettua operaattoria operator==(const String &, const String &)?

```
"cobble" == "stone"
```

Harjoitus 15.2

Tee ylikuormitetut erisuuruusoperaattorit, jotka voivat käsitellä seuraavat kolme tapausta:

```
String != String
String != C-style string
C-style string != String
```

Perustele, miksi valitsit toteutukseen yhden tai useamman operaattorin.

Harjoitus 15.3

Yksilöi, mitkä Screen-luokan jäsenfunktiot, jotka toteutettiin luvussa 13 kohdissa 13.3, 13.4 ja 13.6, ovat hyviä ehdokkaita operaattorin ylikuormitukseen.

Harjoitus 15.4

Selitä, miksi ylikuormitetut syöttö- ja tulostusoperaattorit, jotka määriteltiin kohdassa 3.15 String-luokalle, on esitelty globaaleina funktioina eikä jäsenfunktioina.

Harjoitus 15.5

Toteuta ylikuormitetut syöttö- ja tulostusoperaattorit Screen-luokalle, joka määriteltiin luvussa 13.

15.2 Ystävät

Tutkikaamme uudelleen ylikuormitettujen yhtäsuuruusoperaattorien määrittelyä String-luokalle, joka esiteltiin edellisessä kohdassa. Kahden String-olion yhtäsuuruusoperaattori on seuraavassa:

```
bool operator==( const String &str1, const String &str2 )
{
    if ( str1.size() != str2.size() )
        return false;
    return strcmp( str1.c_str(), str2.c_str() ) ? false : true;
}
```

Vertaa tätä määrittelyä operaattorin määrittelyyn, kun se on määritelty jäsenfunktiona:

```
bool String::operator==( const String &rhs ) const
{
    if ( _size != rhs._size )
        return false;
    return strcmp( _string, rhs._string ) ? false : true;
}
```

Huomaatko eron? Huomaa, että muokkausta tarvittiin sen määräämiseen, kuinka määrittely viittaa String-luokan yksityisiin tietojäseniin. Koska uusi yhtäsuuruusoperaattori on globaali funktio eikä luokan jäsenfunktio, se ei voi viitata String-luokan yksityisiin tietojäseniin suoraan. Sen sijaan se käyttää jäsenen käsittelyfunktioita `size()` ja `c_str()` saadakseen String-olion koon ja sen pohjana olevan C-tyylisen merkkijonon.

Toinen mahdollinen toteutus on esitellä globaalit yhtäsuuruusoperaattorit String-luokan *ystävinä* (*friend*). Kun luokka esittelee funktion tai operaattorin ystävänä, se myöntää tuolle funktiolle tai operaattorille pääsyn sen ei-julkisiin jäseniin.

Ystävän esittely alkaa avainsanalla `friend`. Se voi esiintyä vain luokan määrittelyssä. Koska ystävät eivät ole ystävyyttä myöntävän luokan jäseniä, niihin ei vaikuta julkinen, yksityinen tai suojattu osa, jossa ne esitellään luokan rungossa. Tässä olemme päättäneet ryhmittää kaikki ystäväesittelyt heti luokan otsikon jälkeen:

```
class String {
    friend bool operator==( const String &, const String & );
    friend bool operator==( const char *, const String & );
    friend bool operator==( const String &, const char * );
public:
    // ... loput String-luokasta
};
```

Nuo ystäväesittelyt String-luokassa esittelevät globaalilta viittausalueelta kolme ylikuormitettua vertailuoperaattoria (joihin tutustuttiin edellisessä kohdassa).

Nyt, kun yhtäsuuruusoperaattorit on esitelty ystävinä, voi niiden määrittely viitata String-luokan yksityisiin jäseniin suoraan:

```
// ystäväoperaattorit: viittaavat String-luokan yksityisiin jäseniin suoraan
bool operator==( const String &str1, const String &str2 )
{
    if ( str1._size != str2._size )
        return false;
    return strcmp( str1._string, str2._string ) ? false : true;
}

inline bool operator==( const String &str, const char *s )
{
    return strcmp( str._string, s ) ? false : true;
}
// jne
```

Joku voi onnistuneesti väittää tässä tapauksessa, että _size- ja _string-jäsenten suora käsittely on tarpeetonta, koska c_str() ja size(), jotka ovat välittömiä, ovat yhtä tehokkaita samalla, kun säilyttävät jäsenen kapseloinnin. Käsittelyfunktion käyttö jäsenen suoran käsittelyn sijasta ei aina merkitse tehottomampaa toteutusta. Näiden käsittelyfunktioiden takia ei ole tarvetta esitellä String-luokkamme yhtäsuuruusoperaattoreita ystäväfunktioina.

Kuinka sitten päätämme, pitäisikö operaattori, joka ei ole luokan jäsen, tehdä ystäväksi vai tulisiko sen käyttää jäsenen käsittelyfunktioita? Yleensä luokan toteuttajan tulisi yrittää minimoida nimiavaruuden funktioiden ja operaattoreiden lukumäärää, joilla on pääsy luokan sisäiseen esitystapaan. Jos jäsenen käsittelyfunktiot on tehty ja ne ovat yhtä tehokkaita, on silloin parempi käyttää niitä ja eristää nimiavaruuden operaattorit luokan esitystavan muutoksilta aivan kuten se tehdään muillekin nimiavaruuden funktioille. Jos luokan toteuttaja päättää, että ei tee jäsenen käsittelyfunktioita joillekin luokan yksityisille jäsenille ja jos nimiavaruuden operaattoreiden pitää viitata näihin yksityisiin jäseniin operaatioidensa suorittamiseksi, tulee ystävämekanismin käytöstä tarpeellinen.

Ystäväesittelyiden kaikkein yleisin käyttötilanne on sallia ylikuormitettujen operaattorien, jotka eivät ole luokan jäseniä, pääsy luokan yksityisiin jäseniin, joiden ystäviä ne ovat. Tähän on syynä se, että vasemman ja oikeanpuoleisen operandin symmetrisyyden tekemisen lisäksi ylikuormitetusta operaattorista tulisi muutoin jäsenfunktio täysin oikeuksin luokan yksityisiin jäseniin.

Vaikka ystäväesittelyiden vallitseva käyttötilanne on ylikuormitetut operaattorit, on tilanteita, joissa nimiavaruuden funktio, toisen aikaisemmin määritellyn luokan jäsenfunktio tai koko luokka pitää esitellä ystävänä. Kun luokasta tehdään toisen luokan ystävä, annetaan ystäväluokan jäsenfunktioille pääsy ystävyyden myöntävän luokan yksityisiin jäseniin. Tutkimme tässä tarkemmin ystäväesittelyitä funktioille, jotka eivät ole operaattoreita.

Luokan pitää esitellä ystävänä jokainen sellainen funktio ylikuormitettujen funktioiden

joukossa, josta se haluaa tehdä ystävän. Esimerkiksi:

```
extern ostream& storeOn( ostream &, Screen & );
extern BitMap& storeOn( BitMap &, Screen & );
// ...

class Screen
{
    friend ostream& storeOn( ostream &, Screen & );
    friend BitMap& storeOn( BitMap &, Screen & );
    // ...
};
```

Jos funktio käsittelee kahden erityyppisen luokan olioita ja haluaa käsitellä molempien luokkien ei-julkisia jäseniä, voidaan funktio joko esitellä ystävänä molemmille luokille tai tehdä siitä toisen jäsenfunktio ja ystävä toiselle. Katsotaanpa, kuinka se voitaisiin tehdä.

Jos päätämme, että funktiosta pitää tehdä ystävä molemmille luokille, ovat ystäväesittelyt seuraavanlaisia:

```
class Window; // vain esittely
class Screen {
    friend bool is_equal( Screen &, Window & );
    // ...
};

class Window {
    friend bool is_equal( Screen &, Window & );
    // ...
};
```

Jos päätämme, että funktiosta pitää tehdä toisen luokan jäsenfunktio ja ystävä toiselle luokalle, ovat jäsenfunktion esittely ja ystäväesittely seuraavanlaiset:

```
class Window;
class Screen {
public:
    // copy on Screen-luokan jäsen
    Screen& copy( Window & );
    // ...
};

class Window {
```

```
    // Screen::copy on Window-luokan ystävä
    friend Screen& Screen::copy( Window & );
    // ...
};

Screen& Screen::copy( Window & ) { /* ... */ }
```

Luokan jäsenfunktiota ei voi esitellä toisen luokan ystäväksi ennen kuin sen luokan määrittely on nähty. Tämä ei aina ole mahdollista. Mitä, jos esimerkiksi Screen-luokan pitää esitellä Window-luokan jäsenfunktiot ystävinä ja Window-luokan pitää esitellä Screen-luokan jäsenfunktiot ystävinä? Tapauksissa kuten tässä koko Window-luokka voidaan esitellä Screen-luokan ystävänä. Esimerkiksi:

```
class Window;
class Screen {
    friend class Window;
    // ...
};
```

Screen-luokan ei-julkisia jäseniä voidaan nyt käsitellä jokaisesta Window-jäsenfunktiosta.

Harjoitus 15.6

Toteuta uudelleen Screen-luokalle harjoituksessa 15.5 määritellyt syöttö- ja tulostusoperaattorit ystäväfunktioina ja muokkaa niiden määrittelyjä, jotta ne pääsevät käsittelemään luokan yksityisiä jäseniä suoraan. Kumpi toteutus on parempi? Perustele, miksi.

15.3 Operaattori =

Luokan olion sijoitus toiseen luokkatyyppinsä mukaiseen olioon suoritetaan kopioinnin sijoitusoperaattorilla. Tämä erityinen sijoitusoperaattori kuvattiin kohdassa 14.7.

Luokalle voidaan määritellä muitakin sijoitusoperaattoreita. Jos luokkatyypin olioihin aiotaan sijoittaa arvoja, jotka ovat muuta kuin niiden omaa tyyppiä, voidaan määritellä sijoitusoperaattoreita, jotka saavat parametrina tuon toisen tyypin. Jotta voisimme tukea esimerkiksi C-tyylisen merkkijonon sijoitusta String-luokkamme olioon kuten tässä

```
String car ("Volks");
car = "Studebaker";
```

teemme seuraavan sijoitusoperaattorin, joka hyväksyy tyypin `const char*`. Tämä operaattori on esitelty aikaisemmin String-luokassamme:

```
class String {
public:
    // sijoitusoperaattori char*-tyypille
    String& operator=( const char * );
    // ....
private:
    int   _size;
    char *_string;
};
```

Tämä sijoitusoperaattori on toteutettu seuraavasti. Jos String-olioon sijoitetaan null-osoittimen arvo, asetetaan String-olio tyhjäksi; muussa tapauksessa String-olion sisältö on C-tyylisen merkkijonon kopio, joka siihen on sijoitettu:

```
String& String::operator=( const char *sobj )
{
    // sobj on null-osoitin
    if ( ! sobj ) {
        _size = 0;
        delete[] _string;
        _string = 0;
    }
    else {
        _size = strlen( sobj );
        delete[] _string;
        _string = new char[ _size + 1 ];
        strcpy( _string, sobj );
    }
    return *this;
}
```

_string viittaa C-tyylisen merkkijonon kopioon, johon sobj-parametri viittaa. Miksi kopio? Emme voi sijoittaa sobj-parametria _string-olioon suoraan:

```
_string = sobj; // virhe: tyyppivirhe
```

sobj on osoitin const:iin. Osoitinta const:iin ei voi sijoittaa osoittimeen ei-const:iin (kuten kuvattiin kohdassa 3.5). Voisimme päättää määritellä sijoitusoperaattorin kuten seuraavassa

```
String& String::operator=( char *sobj ) { // ...
```

joka nyt mahdollistaa, että _string voi viitata suoraan C-tyyliseen merkkijonoon, johon sobj viittaa. Tämä kuitenkin luo muita ongelmia. Muista, että C-tyylisen merkkijonon tyyppi on const char*. Parametrin määrittely osoittimeksi ei-const:iin estää sijoituslausekkeen, jonka halusimme mahdollistaa alun perin!

```
car = "Studebaker"; // ei sallittu operaattorilla operator=( char * ) !
```

Meillä ei ole valinnanvaraa. Parametrin pitää olla const char*, jotta voisimme sallia C-tyylisten merkkijonojen sijoituksen String-olioihin. Tulee muita ongelmia, jos _string on tehty viittaamaan suoraan C-tyyliseen merkkijonoon, johon sobj viittaa. Emme tiedä, mihin sobj viittaa.

Se voi viitata merkkijonoon, jota voidaan muokata String-oliolle tuntemattomalla tavalla. Esimerkiksi:

```
char ia[] = { 'd', 'a', 'n', 'c', 'e', 'r' };
String trap = ia; // trap._string viittaa ia:han
ia[3] = 'g';  // ei sitä, mitä haluamme: muuttaa ia:ta ja trap._string:iä
```

Jos trap._string tehtäisiin viittaamaan suoraan ia:han, voisi trap-olio käyttäytyä yllättävällä tavalla; sen arvoa voitaisiin muuttaa käynnistämättä yhtään String-jäsenfunktiota. Sen vuoksi uskomme, että muistin varaus, joka voisi sisältää C-tyylisen merkkijonon arvon, johon _string viittaa, voisi saada String-luokkamme olioiden käyttäytymisen yllätyksettömämmäksi.

Huomaa, että sijoitusoperaattorimme käyttää delete-lauseketta. _string viittaa merkkitaulukkoon, joka on varattu keosta. Jotta voisimme estää muistin loppumisen, C-tyylinen merkkijono, johon _string viittaa, vapautetaan käyttämällä delete-lauseketta ennen kuin _string saadaan viittaamaan varattuun muistialueeseen, joka voi sisältää uuden merkkijonon arvon. Koska _string viittaa merkkitaulukkoon, pitää delete-lausekkeesta käyttää taulukkoversiota. (delete-lausekkeen taulukkoversiota käsitellään kohdassa 8.4.)

Eräs viimeisistä asioista, joka koskee sijoitusoperaattoria: sijoitusoperaattorin paluutyyppi on viittaus String-luokkaan. Miksi esittelisimme sijoitusoperaattorin palauttamaan viittauksen? Sisäisillä tyypeillä sijoitusoperaattorit voidaan ketjuttaa kuten seuraavassa:

```
// sijoitusoperaattorien ketju
int iobj, jobj;
iobj = jobj = 63;
```

Sijoitusoperaattorit assosioivat oikealta vasemmalle. Edellisten sijoitusten järjestys on seuraava:

```
iobj = (jobj = 63);
```

Haluaisimme säilyttää tämän piirteen String-luokkamme olioiden sijoituksiin niin, että tuetaan sijoitusta kuten seuraavassa:

```
String verb, noun;
verb = noun = "count";
```

Ensimmäinen sijoitus tässä ketjussa kutsuu aiemmin const char* -tyypille määriteltyä sijoitusoperaattoria. Tämän sijoituksen tuloksen tyypin pitää olla sellainen, että sitä voidaan käyttää argumenttina String-luokan kopioinnin sijoitusoperaattorille. Tästä syystä, vaikka sijoitusoperaattoriparametri onkin const char*, sen paluutyyppi on viittaus String-luokkaan.

Sijoitusoperaattoreita voidaan ylikuormittaa. String-luokassamme ylikuormitettujen sijoitusoperaattorien joukko on seuraava:

```
// sijoitusoperaattorien ylikuormitettu joukko
String& operator=( const String & );
String& operator=( const char * );
```

Jokaiselle tyypille, joka pitää sijoittaa String-olioon, voi olla sijoitusoperaattori. Jokainen sijoitusoperaattori pitää kuitenkin määritellä luokan jäsenfunktiona.

15.4 Operaattori [ ]

Indeksioperaattori operator[]() voidaan määritellä luokille, jotka edustavat säiliöabstraktiota ja joista haetaan yksittäisiä elementtejä. String-luokkamme, luvussa 2 esitelty IntArray-luokka tai C++-vakiokirjastossa määritelty mallivektoriluokka, ovat esimerkkejä säiliöluokista, joille on järkeä esitellä indeksioperaattori. Indeksioperaattori pitää määritellä luokan jäsenfunktioksi.

String-luokkamme käyttäjillä pitää olla sekä luku- että kirjoitusoikeus _string-luokkajäsenen yksittäisiin merkkeihin. Haluamme tukea seuraavaa String-luokkaolioiden käyttötapaa:

```
String entry( "extravagant" );
String mycopy;

for ( int ix = 0; ix < entry.size(); ++ix )
    mycopy[ ix ] = entry[ ix ];
```

Indeksioperaattorin pitää kyetä esiintymään sijoitusoperaattorin sekä oikealla että vasemmalla puolella. Jotta se voisi esiintyä vasemmalla puolella, sen paluuarvon pitää olla *lvalue*. Tämä saavutetaan määrittämällä paluutyyppi viittaukseksi:

```
#include <cassert>

inline char&
String::operator[]( int elem ) const
{
    assert( elem >= 0 && elem < _size );
    return _string[ elem ];
}
```

Indeksioperaattorin paluutyyppi on indeksielementin lvalue. Tästä syystä se voi esiintyä sijoituksen vasemmalla puolella. Esimerkiksi seuraavassa sijoitetaan merkkiarvo nollaelementtiseen color._string:iin:

```
String color( "violet" );
color[0] = 'V';
```

Huomaa, että indeksioperaattori on määritelty suorittamaan rajatarkistus vastaanottamalleen indeksille. Tässä päätimme käyttää C-kirjastofunktiota assert(), joka suorittaa tämän tarkistuksen. Voisimme sen asemesta heittää poikkeuksen ja ilmaista, että elem-indeksin arvo on negatiivinen tai suurempi kuin C-tyylinen merkkijono, johon _string viittaa. (Poikkeuskäsittelyn throw-lausekkeet on käsitelty luvussa 11.)

15.5 Operaattori ( )

Funktion kutsuoperaattoria voidaan ylikuormittaa luokkatyyppisille oliolle. Olemme jo nähneet tämän ylikuormitetun operaattorin, kun käsittelimme funktio-olioita kohdassa 12.3. Jos luokkatyyppi on määritelty edustamaan operaatiota, voidaan tämän luokkatyypin funktion kutsuoperaattori käynnistää tuolle operaatiolle. Esimerkiksi absInt-luokka on määritelty kapseloimaan operaatio, jossa asetetaan sen int-tyyppiin parametrin absoluuttinen arvo:

```
class absInt {
public:
    int operator()( int val ) {
        int result = val < 0 ? -val : val;
        return result;
    }
};
```

Ylikuormitettu operator()-operaattori pitää esitellä jäsenfunktiona. Sen parametriluettelossa voi olla kuinka monta parametria tahansa. Parametrien tyypit voivat olla mitä tahansa tyyppejä, jotka ovat sallittuja kohdissa 7.2 ja 7.3 esitetyissä funktioparametreissa. Ylikuormitetun operator()-operaattorin paluuarvo voi olla mikä tahansa tyyppi, joka on sallittu kohdissa 7.2 ja 7.4 esitetyissä funktion paluuarvoissa.

Ylikuormitettu operator()-operaattori käynnistetään käyttämällä argumenttiluetteloa luokkatyyppinsä olioon. Tulemme tutkimaan, kuinka luvussa 12 määritelty geneerinen algoritmi käyttää absInt-luokkamme ylikuormitettua operator()-operaattoria. Seuraavassa esimerkissä kutsutaan geneeristä transform()-algoritmia, joka käyttää absInt-luokassa määriteltyä operaatiota jokaiselle ivec-vektorin sisältämälle elementille; tarkoittaa, että se asettaa jokaisen vektorin elementin sen absoluuttiseen arvoon.

```
#include <vector>
#include <algorithm>

int main() {
    int ia[] = { -0, 1, -1, -2, 3, 5, -5, 8 };
    vector< int > ivec( ia, ia+8 );

    // aseta jokainen ivec:in elementti absoluuttiseen arvoonsa
    transform( ivec.begin(), ivec.end(), ivec.begin(), absInt() );

    // ....
}
```

transform()-algoritmin ensimmäinen ja toinen argumentti ilmaisevat elementtialueen, johon absInt-operaatiota käytetään. Kolmas argumentti viittaa vektorin alkuun, johon absInt-operaation tulos tallennetaan.

transform()-algoritmin neljäs argumentti on tilapäinen absInt-luokan olio, joka on luotu käynnistämällä absInt-luokan oletusmuodostaja. Geneerisen transform()-algoritmin instantioin-

nin käynnistys main()-funktiossa voisi näyttää tältä:

```
typedef vector<int>::iterator iter_type;

// transform():in instantiointi
// käyttää absInt-operaatiota
// int-tyyppisen vektorin elementteihin

iter_type transform( iter_type iter, iter_type last,
                     iter_type result, absInt func )
{
    while ( iter != last )
        *result++ = func( *iter++ ); // käynnistää: absInt::operator()

    return iter;
}
```

func on luokkatyyppinen olio ja edustaa absInt-operaatiota, joka asettaa int-tyypin arvon absoluuttiseen arvoonsa. func-oliota käytetään absInt-luokan ylikuormitetun operator()-operaattorin käynnistykseen. Tälle ylikuormitetulle operaattorille välitetty operaattori on *iter, joka viittaa vektorin elementtiin, johon haluamme saada absoluuttisen arvon.

15.6 Operaattori ->

Jäsenen käsittelyoperaattoria, nuolta, voidaan ylikuormittaa luokkatyyppisille oliolle. Se pitää määritellä luokan jäsenfunktiona. Se on määritelty antamaan luokkatyypille ‘‘osoitintyyppistä’’ käyttäytymistä. Sitä käytetään useimmin luokkatyypin kanssa, joka edustaa ‘‘fiksua osoitinta’’; tarkoittaa luokkaa, joka käyttäytyy melko samalla tavalla kuin sisäinen osoitintyyppi, mutta tukee joitakin lisätoimintoja.

Oletetaan esimerkiksi, että haluamme määritellä luokkatyypin, joka edustaa osoitinta Screen-luokkaolioon, jossa Screen-luokka on esitelty luvussa 13:

```
class ScreenPtr {
    // ...
private:
    Screen *ptr;
};
```

Haluamme määritellä ScreenPtr-luokan niin, että tämän tyyppisen olion taataan aina viittaavan Screen-olioon. Se ei voi olla viittaamatta mihinkään olioon, kuten sisäinen osoitin. Sovelluksemme voivat sitten käyttää ScreenPtr-tyyppisiä olioita tarvitsematta testata ensin, viittaavatko ne Screen-olioon vai eivät. Jotta saavuttaisimme tämän piirteen, määrittelemme ScreenPtr-luokkaan muodostajan, mutta emme oletusmuodostajaa (muodostaja on käsitelty yksityiskohtaisesti kohdassa 14.2):

```
class ScreenPtr {
public:
    ScreenPtr( const Screen &s ) : ptr( &s ) { }
```

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

ScreenPtr-tyyppisen olion määrittelyssä pitää olla alustaja, Screen-tyyppinen olio, johon ScreenPtr-olio laitetaan viittaamaan, muussa tapauksessa ScreenPtr-olion määrittely on virheellinen:

```
ScreenPtr p1; // virhe: ScreenPtr:llä ei ole oletusmuodostajaa

Screen myScreen( 4, 4 );
ScreenPtr ps( myScreen ); // ok
```

Jotta ScreenPtr-luokka käyttäytyisi kuten sisäinen osoitin, pitää määritellä joitakin ylikuormitettuja operaattoreita. Kaksi operaattoria, jotka määrittelemme, ovat käänteisoperaattori (*) ja jäsenen käsittelyn nuolioperaattori:

```
// ylikuormitetut operaatiot tukemaan osoittimen käyttäytymistä
class ScreenPtr {
public:
    Screen& operator*()  { return *ptr; }
    Screen* operator->() { return ptr; }
    //....
};
```

Jäsenen käsittelyn nuolioperaattori on ylikuormitettu unaarisena; tarkoittaa, että se ei saa parametria. Kun sitä käytetään lausekkeessa, se valitaan ainoastaan vasemman operandin tyypin perusteella. Esimerkiksi lauseessa

```
point->action();
```

tutkitaan point sen tyypin päättelemiseksi. Jos point on jonkin luokkatyypin osoitin, lause käyttää sisäistä käsittelyn nuolioperaattorin merkitystä. Jos point on olio tai jonkin luokkatyypin viittaus, luokasta etsitään ylikuormitettua jäsenen käsittelyn nuolioperaattoria. Ellei jäsenen nuolioperaattoria ole määritelty, lause on virheellinen, koska luokkaolion tai viittauksen pitää tavallisesti käyttää jäsenen käsittelyn pisteoperaattoria luokan jäseniin viittauksiin. Jos ylikuormitettua jäsenen käsittelyn nuolioperaattoria ei ole määritelty, se sidotaan point:iin ja käynnistetään.

Ylikuormitetun jäsenen käsittelyn nuolioperaattorin paluutyypin pitää olla joko osoitin luokkatyyppiin tai luokan olio, josta jäsenen käsittelyn nuolioperaattori on määritelty. Jos paluutyyppi on osoitin luokkatyyppiin, käytetään sisäistä jäsenen käsittelyn nuolioperaattoria arvon palauttamiseen. Jos paluuarvo on luokkaolio tai viittaus, prosessia käytetään rekursiivisesti, kunnes palautetaan joko osoitintyyppi tai todetaan lauseen olevan virheellinen. Voimme esimerkiksi käyttää ScreenPtr-oliota ps käsitelläksemme Screen-luokan jäseniä kuten seuraavassa:

```
ps->move( 2, 3);
```

Koska jäsenen nuolikäsittelyoperaattorin vasen operandi on ScreenPtr-luokkatyyppinen, käytetään tämän luokan ylikuormitettua operaattoria. Operaattori palauttaa osoittimen Screen-

luokkaolioon. Sisäistä jäsenen käsittelyn nuolioperaattoria käytetään vuorostaan tälle paluuarvolle, joka kutsuu Screen-luokan move()-jäsenfunktiota.

Seuraavassa on pieni ohjelma, joka kokeilee ScreenPtr-luokkaamme. ScreenPtr-tyyppistä oliota käytetään aivan kuin mitä tahansa Screen*-tyyppistä oliota:

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

void printScreen( const ScreenPtr &ps )
{
    cout << "Screen Object ("
       << ps->height() << ", "
       << ps->width() << " )\n\n";

    for ( int ix = 1; ix <= ps->width(); ++ix )
    {
        for ( int iy = 1; iy <= ps->height(); ++iy )
            cout << ps->get( ix, iy );
        cout << "\n";
    }
}

int main() {
    Screen sobj( 2, 5 );
    string init( "HelloWorld" );
    ScreenPtr ps( sobj );

    // Aseta näytön sisältö
    string::size_type initpos = 0;
    for ( int ix = 1; ix <= ps->width(); ++ix )
        for ( int iy = 1; iy <= ps->height(); ++iy )
        {
            ps->move( ix, iy );
            ps->set( init[ initpos++ ] );
        }

    // Tulosta näytö sisältö
    printScreen( ps );

    return 0;
}
```

Tietystikään tämänkaltainen osoitinten käsittely luokkaolioihin ei ole yhtä tehokasta kuin sisäisten osoitintyyppien käyttö. Fiksussa osoitinluokassa pitää siten olla lisätoimintoja, jotka kompensoivat luokan käytön aiheuttamaa lisäkuormaa. Tämä on tärkeää sovelluksesi suunnittelussa.

15.7 Operaattorit ++ ja --

Jatkakaamme edellisessä kohdassa esitellyn ScreenPtr-luokan toteutusta. On olemassa kaksi muuta sisäisille osoitintyypeille tuettua operaattoria, jotka haluaisimme määritellä tälle luokalle: kasvatus (++)- ja vähennys (--) -operaattorit. Haluaisimme kyetä käyttämään ScreenPtr-luokkaamme niin, että sillä voitaisiin viitata Screen-oliotaulukon elementteihin. Jotta voimme sen tehdä, pitää lisätä joitakin tietojäseniä ScreenPtr-luokkaamme.

Ensiksi määrittelemme uuden tietojäsenen nimeltään size, joka sisältää joko nolla-arvon (ilmaisee, että ScreenPtr-olio osoittaa yksittäiseen olioon) tai taulukon koon, johon ScreenPtr-olio viittaa. Määrittelemme myös tietojäsenen nimeltään offset, joka muistaa siirtymän taulukossa, johon ScreenPtr-olio viittaa:

```
class ScreenPtr {
public:
    // ...
private:
    int size;    // taulukon koko; nolla, jos yksittäinen olio
    int offset;  // ptr:n siirtymä taulukossa
    Screen *ptr;
};
```

Tällä lisätoiminnolla ja näillä uusilla tietojäsenillä pitää ScreenPtr-luokan muodostajaa muokata. ScreenPtr-luokkamme käyttäjän pitää antaa lisäargumentti muodostajalle, jos luotu ScreenPtr-olio viittaa taulukkoon:

```
class ScreenPtr {
public:
    ScreenPtr( const Screen &s , int arraySize = 0 )
        : ptr( &s ), size ( arraySize ), offset( 0 ) { }
private:
    int size;
    int offset;
    Screen *ptr;
};
```

Muodostajan lisäargumentti ilmaisee taulukon koon. Jotta säilyttäisimme aikaisemman toiminnallisuuden, asetetaan muodostajan toiseen parametriin oletusargumentti ja size:n arvoksi nolla, jos toista argumenttia ei anneta ScreenPtr-luokkaa luotaessa. Tässä tapauksessa on oletettu, että olio viittaa yhteen Screen-olioon. Uuden ScreenPtr-luokkamme oliot voidaan määritellä seuraavasti:

```
Screen myScreen( 4, 4 );
ScreenPtr pobj( myScreen ); // ok: viittaa yhteen olioon

const int arrSize = 10;
Screen *parray = new Screen[ arrSize ];
ScreenPtr parr( *parray, arrSize ); // ok: viittaa taulukkoon
```

Olemme nyt valmiit määrittelemään ylikuormitetut kasvatus- ja vähennysoperaattorit ScreenPtr-luokalle. On kuitenkin pieni ongelma. On olemassa kahdenlaisia kasvatus- ja vähennysoperaattoreita: etuliite- ja jälkiliiteversiot. Onneksi sekä etuliite- että jälkiliiteilmentymät ylikuormitetuista kasvatus- ja vähennysoperaattoreista voidaan määritellä. Etuliiteoperaattoreiden esittelyt voisivat näyttää odotetusti tältä:

```
class ScreenPtr {
public:
    Screen& operator++();
    Screen& operator--();
    // ...
};
```

Kasvatus- ja vähennysoperaattorien jälkiliiteilmentymät on määritelty unaarisina operaattorifunktioina. Esimerkiksi etuliiteoperaattoria voidaan käyttää kuten seuraavassa:

```
const int arrSize = 10;
Screen *parray = new Screen[ arrSize ];
ScreenPtr parr( *parray, arrSize );

for ( int ix = 0;
    ix < arrSize;
    ++ix, ++parr // yhtä kuin parr.operator++()
  )
        printScreen( parr );
```

Nämä ylikuormitetut operaattorit voidaan määritellä seuraavasti:

```
Screen& ScreenPtr::operator++()
{
    if ( size == 0 ) {
        cerr << "cannot increment pointer to single object\n";
        return *ptr;
    }
    if ( offset >= size - 1 ) {
        cerr << "already at the end of the array\n";
        return *ptr;
    }

    ++offset;
    return *++ptr;
}

Screen& ScreenPtr::operator--()
{
    if ( size == 0 ) {
        cerr << "cannot decrement pointer to single object\n";
        return *ptr;
    }
    if ( offset <= 0 ) {
```

```
            cerr << "already at the beginning of the array\n";
            return *ptr;
        }

        --offset;
        return *--ptr;
    }
```

Jotta voitaisiin erottaa jälkiliiteoperaattorin esittely etuliiteoperaattorin esittelystä, on kasvatus- ja vähennysoperaattorien jälkiliiteilmentymän esittelyihin lisätty ylimääräinen parametri, joka on int-tyyppinen. Seuraavassa esimerkissä on esitelty ScreenPtr-luokkaan etuliite- ja jälkiliiteoperaattoriparit.

```
class ScreenPtr {
public:
    Screen& operator++();    // etuliiteoperaattorit
    Screen& operator--();
    Screen& operator++(int); // jälkiliiteoperaattorit
    Screen& operator--(int);
    // ...
};
```

Jälkiliiteoperaattorit voitaisiin toteuttaa seuraavasti:

```
Screen& ScreenPtr::operator++(int)
{
    if ( size == 0 ) {
        cerr << "cannot increment pointer to single object\n";
        return *ptr;
    }
    if ( offset == size ) {
        cerr << "already one past the end of the array\n";
        return *ptr;
    }

    ++offset;
    return *ptr++;
}

Screen& ScreenPtr::operator--(int)
{
    if ( size == 0 ) {
        cerr << "cannot decrement pointer to single object\n";
        return *ptr;
    }
    if ( offset == -1 ) {
        cerr << "already one before the beginning of the array\n";
        return *ptr;
    }
```

```
        --offset;
        return *ptr--;
    }
```

Huomaa, että parametrille ei tarvitse antaa nimeä, koska sitä ei käytetä operaattorin määrittelyssä. Kokonaisluvun lisäparametri on läpinäkyvä jälkiliiteoperaattorin käyttäjille. Kääntäjä antaa sille oletusarvon, joka voidaan jättää huomiotta. Tästä syystä parametri on jätetty nimeämättä. Tässä on esimerkki jälkiliiteoperaattorin käyttötilanteesta:

```
const int arrSize = 10;
Screen *parray = new Screen[ arrSize ];
ScreenPtr parr( *parray, arrSize );

for ( int ix = 0; ix < arrSize; ++ix )
    printScreen( parr++ );
```

Eksplisiittinen kutsu jälkiliiteoperaattoriin vaatii, että toisen argumentin kokonaisluvulle määritetään todellinen arvo. ScreenPtr-luokassamme jätetään huomiotta tämän eksplisiittisen kutsun argumentti, koska sitä ei käytetä ylikuormitetun operaattorin määrittelyssä:

```
parr.operator++(1024); // kutsu jälkiliiteoperaattoria operator++
```

Ylikuormitetut kasvatus- ja vähennysoperaattorit voidaan esitellä myös ystäväfunktioina. Voimme esimerkiksi muuttaa ScreenPtr-luokan määrittelyä niin, että se esittelee nämä operaattorit ystäväfunktioina kuten seuraavassa:

```
class ScreenPtr {
    // muiden kuin jäsenten esittelyt
    friend Screen& operator++( ScreenPtr & );      // etuliite
    friend Screen& operator--( ScreenPtr & );
    friend Screen& operator++( ScreenPtr &, int ); // jälkiliite
    friend Screen& operator--( ScreenPtr &, int );
public:
    // jäsenten määrittelyt
};
```

Harjoitus 15.7

Tee määrittelyt ScreenPtr-luokan ylikuormitetuille kasvatus- ja vähennysoperaattoreille, kun ne on esitelty ystäväfunktioina.

Harjoitus 15.8

ScreenPtr-luokka voi nyt edustaa osoitinta, joka osoittaa Screen-luokkien taulukkoon. Muokkaa ylikuormitettua operator*()-operaattoria ja operator->()-operaattoria (määritelty kohdassa 15.6) varmistaaksesi, että jos ScreenPtr-olio viittaa taulukon elementtiin, osoitin ei viittaa yhtä ennen taulukon alkua eikä yhtä yli taulukon lopun. Vinkki: näiden ylikuormitettujen operaattoreiden tulisi käyttää uusia size- ja offset-tietojäseniä.

15.8 Operaattorit new ja delete

Oletusarvo on, että luokkaolioiden varaaminen ja vapauttaminen vapaasta varastosta suoritetaan globaaleilla new()- ja delete()-operaattoreilla, jotka on määritelty C++-vakiokirjastossa. (Esittelimme nämä operaattorit kohdassa 8.4.) Luokka voi hoitaa oman muistinhallinnan tekemällä luokan jäsenoperaattorit nimeltään new() ja delete(). Jos nämä operaattorit on määritelty luokkaan, ne käynnistetään globaalien operaattorien sijasta varaamaan ja vapauttamaan luokkatyyppinsä mukaisia olioita. Määritelkäämme esimerkkinä new()- ja delete()-operaattorit Screen-luokkamme jäseninä.

Luokan new()-jäsenoperaattorin paluutyypin pitää olla void* ja sen ensimmäisen parametrin on oltava tyyppiä size_t, joka on järjestelmäotsikkotiedostoon <cstddef> määritelty typedef-nimi. Tässä on Screen-luokan new()-operaattorin esittely:

```
class Screen {
public:
    void *operator new( size_t );
    // ...
};
```

Kun new-lauseke luo luokkatyyppisen olion, kääntäjä etsii nähdäkseen, onko luokalla new()-jäsenoperaattoria. Jos sillä on, tämä operaattori valitaan varaamaan muistia luokkaoliolle; muussa tapauksessa kutsutaan globaalia new()-operaattoria. Esimerkiksi seuraava new-lauseke

```
Screen *ps = new Screen;
```

luo Screen-luokkatyyppisen olion vapaavarastoon ja koska Screen-luokalla on new()-jäsenoperaattori, kutsutaan sitä. Operaattorin size_t-parametri alustetaan automaattisesti arvolla, joka edustaa Screen-luokan kokoa tavuina.

new()-operaattorin lisääminen tai poistaminen luokasta ei vaadi muutoksia käyttäjän koodiin. new-lausekkeen muoto on sama, kutsuu se sitten globaalia new()-operaattoria tai luokan new()-jäsenoperaattoria. Jos Screen-luokka ei määrittele omaa new()-operaattoriaan, on new-lauseke silti kelvollinen, mutta kutsuu sen sijasta globaalia new()-operaattoria.

Ohjelmoija voi käynnistää valikoivasti globaalin new()-operaattorin käyttämällä globaalin viittausalueen erotteluoperaattoria. Esimerkiksi

```
Screen *ps = ::new Screen;
```

käynnistää globaalin new()-operaattorin, vaikka Screen-luokassa on määritelty new()-jäsenoperaattori.

Luokan delete()-jäsenoperaattorin paluutyypin pitää olla void ja sen ensimmäisen parametrin tyyppiä void*. Tässä on Screen-luokkamme delete()-operaattorin esittely:

```
class Screen {
public:
    void operator delete( void * );
};
```

Kun delete-lausekkeen operaattori on osoitin luokkatyyppiseen olioon, kääntäjä etsii nähdäkseen, onko luokalla delete()-jäsenoperaattori. Jos sillä on, valitaan tämä operaattori vapauttamaan luokkaolion muisti; muussa tapauksessa kutsutaan globaalia delete()-operaattoria. Seuraava delete-lauseke

```
delete ps;
```

vapauttaa Screen-luokan olion muistin, johon ps viittaa. Koska Screen-luokalla on delete()-jäsenoperaattori, kutsutaan sitä. Operaattorin void*-parametri alustetaan automaattisesti ps:n arvolla.

delete()-operaattorin lisääminen tai poistaminen ei vaadi muutoksia käyttäjän koodiin. delete-lausekkeen muoto on sama, kutsuu se sitten globaalia delete()-operaattoria tai luokan delete()-jäsenoperaattoria. Jos Screen-luokka ei määrittele omaa delete()-operaattoriaan, on delete-lauseke silti kelvollinen, mutta kutsuu sen sijasta globaalia delete()-operaattoria.

Ohjelmoija voi valikoivasti käynnistää globaalin delete()-operaattorin käyttämällä globaalin viittausalueen erotteluoperaattoria. Esimerkiksi:

```
::delete ps;
```

käynnistää delete()-operaattorin, joka on määritelty globaalille viittausalueelle, vaikka Screen-luokassa on määritelty delete()-jäsenoperaattori. Yleensä käytettyä delete()-operaattoria tulisi vastata new()-operaattori, jota käytettiin muistin varaamiseen. Jos esimerkiksi ps viittaa muistialueeseen, joka oli varattu new-lausekkeella, joka käynnisti globaalin new()-operaattorin, silloin delete-lausekkeen tulisi myös käynnistää globaali delete()-operaattori.

Luokkatyypille määritelty delete()-operaattori delete-lausekkeen kutsumana voi sisältää kaksi parametria yhden sijasta. Ensimmäisen parametrin pitää silti olla tyyppiä void*. Toisen parametrin tulee olla esimääriteltyä size_t-tyyppiä (muista ottaa mukaan kirjaston otsikkotiedosto <cstddef>). Esimerkiksi:

```
class Screen {
public:
    // korvaa:
    // void operator delete( void * );
    void operator delete( void *, size_t );
};
```

Jos lisäparametri on mukana, kääntäjä alustaa sen automaattisesti ensimmäisen parametrin osoittaman olion koolla tavuina. (Tämä parametri on olennainen oliosuuntautuneessa luokkahierarkiassa, jossa johdettu luokka voi periä delete()-operaattorin. Luvussa 17 käsitellään periytyminen tarkemmin.)

Tutkikaamme tarkemmin Screen-luokkamme new()- ja delete()-operaattorien toteutusta. Muistinvarausstrategiamme on käyttää linkitettyä listaa Screen-luokan olioille, johon freeStore-osoitin osoittaa. Jokainen kutsu Screen-luokan new()-jäsenoperaattoriin palauttaa seuraavan freeStore-osoittimen osoittaman luokkaolion. Jokainen kutsu delete()-jäsenoperaattoriin palauttaa luokkaolion, joka sijaitsee freeStore:n osoittaman listan alussa. Jos freeStore:n osoittama luokkaolioiden linkitetty lista on tyhjä, tehdään kutsu globaaliin new()-operaattoriin, joka varaa muis-

tilohkon, johon mahtuu screenChunk-määrä Screen-oliota.

Sekä screenChunk että freeStore sisältävät arvoja, jotka kiinnostavat vain Screen-luokkaa. Tästä syystä haluamme kapseloida ne Screen-luokan yksityisiksi jäseniksi. Lisäksi näistä tietojäsenistä saa olla vain yksi ilmentymä kaikille Screen-luokasta luoduille oliolle. Tästä syystä nämä jäsenet on esitelty staattisina jäseninä. Kolmas tietojäsen, next, on määritelty pitämään kunnossa Screen-olioiden linkitettyä listaa.

```
class Screen {
public:
    void *operator new( size_t );
    void operator delete( void *, size_t );
    // ...
private:
    Screen *next;
    static Screen *freeStore;
    static const int screenChunk;
};
```

Tässä on eräs mahdollinen toteutus Screen-luokan new()-jäsenoperaattorista:

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

// staattiset jäsenet alustetaan
// ohjelmatekstitiedostoissa, ei otsikkotiedostoissa
Screen *Screen::freeStore = 0;
const int Screen::screenChunk = 24;

void *Screen::operator new( size_t size )
{
    Screen *p;

    if ( !freeStore ) {
        // linkitetty lista tyhjä: kaappaa muistilohko
        // tämä kutsu kohdistuu globaaliin new-operaattoriin
        size_t chunk = screenChunk * size;
        freeStore = p =
            reinterpret_cast< Screen* >( new char[ chunk ] );

        // käsittele nyt varattu muisti
        for ( ;
            p != &freeStore[ screenChunk - 1 ];
            ++p )
            p->next = p+1;
        p->next = 0;
    }

    p = freeStore;
    freeStore = freeStore->next;
    return p;
```

```
}
```

Tässä on eräs mahdollinen toteutus Screen-luokan delete()-jäsenoperaattorista:

```
void Screen::operator delete( void *p, size_t )
{
    // lisää "poistettu" olio takaisin
    // vapaaseen listaan

    ( static_cast< Screen* >( p ) )->next = freeStore;
    freeStore = static_cast< Screen* >( p );
}
```

On mahdollista esitellä luokalle new()-operaattori esittelemättä vastaavaa delete()-operaattoria. Tässä tapauksessa luokkaoliot poistetaan käyttäen globaalia delete()-operaattoria. On myös mahdollista esitellä delete()-operaattori esittelemättä vastaavaa new()-operaattoria. Tässä tapauksessa luokkaoliot luodaan käyttäen globaalia new()-operaattoria. Nämä operaattorit tulevat kuitenkin tavallisesti pareina ja, kuten tässä esimerkissä, luokan suunnittelija tarvitsee niitä useimmin molempia.

new()- ja delete()-operaattorit ovat luokkansa staattisia jäseniä ja ne säilyttävät tavalliset staattisten funktioiden rajoitukset. Nämä operaattorit on tehty staattisiksi jäsenfunktioiksi niin, ettei ohjelmoijan tarvitse esitellä niitä eksplisiittisesti staattisiksi. Muista erityisesti, että staattisessa jäsenfunktiossa ei ole this-osoitinta ja se voi siksi käsitellä vain luokkansa staattisia tietojäseniä suoraan. (Katso kohtaa 13.5, jossa käsitellään staattisia jäsenfunktioita.) Syy tähän on se, että nämä operaattorit käynnistetään joko ennen luokkaolion muodostamista (new()-operaattori) tai sen jälkeen, kun se on tuhottu (delete()-operattori).

Muistin varaus new()-operaattoria käyttäen kuten tässä

```
Screen *ptr = new Screen( 10, 20 );
```

on yhtäpitävä seuraaville kahdelle lauseelle:

```
// C++-pseudokoodia
ptr = Screen::operator new( sizeof( Screen ) );
Screen::Screen( ptr, 10, 20 );
```

Tämä tarkoittaa, että new-lauseke kutsuu ensin luokan new()-operaattoria muistin varaamista varten ja sitten muodostajaa olion alustamista varten. Jos new()-operaattori epäonnistuu, heitetään bad_alloc-tyyppinen poikkeus eikä muodostajaa koskaan käynnistetä.

Muistin vapautus delete()-operaattorilla kuten tässä

```
delete ptr;
```

on yhtäpitävä seuraaville kahdelle lauseelle:

```
// C++-pseudokoodia
Screen::~Screen( ptr );
Screen::operator delete( ptr, sizeof( *ptr ) );
```

Tarkoitaa, että delete-lauseke kutsuu ensin olion tuhoajaa ja sitten luokan delete()-operaattoria muistin vapauttamiseen. Jos ptr:n arvo on 0, ei tuhoajaa eikä delete()-operaattoria koskaan kutsuta.

15.8.1 Taulukko-operaattorit new[] ja delete[]

Luokkamme new()-operaattori, joka määriteltiin edellisessä alikohdassa, käynnistetään vain yksittäisten luokkaolioiden muistin varaamiseen. Esimerkiksi seuraava new-lauseke käynnistää Screen-luokan new()-operaattorin:

```
// käynnistää Screen::operator new()
Screen *ps = new Screen( 24, 80 );
```

kun taas seuraava new-lauseke käynnistää globaalin new[]()-operaattorin, joka käsittelee Screen-olioiden muistinvarauksen vapaavarastosta:

```
// käynnistää ::operator new[]()
Screen *psa = new Screen[10];
```

On myös mahdollista esitellä new[]()- ja delete[]()-operaattorit taulukoille, jotka ovat luokan jäseniä.

Luokan new[]()-jäsenoperaattorin paluutyypin pitää olla void* ja ensimmäisen parametrin size_t-tyyppinen. Tässä on Screen-luokkamme new[]()-operaattorin esittely:

```
class Screen {
public:
    void *operator new[]( size_t );
    // ...
};
```

Kun new-lauseke luo luokkatyypisten olioiden taulukon, kääntäjä etsii luokasta new[]()-jäsenoperaattoria. Jos se löytyy, valitaan tämä operaattori muistin varaamiseen taulukolle; muussa tapauksessa kutsutaan globaalia new[]()-operaattoria. Seuraava new-lauseke luo kymmenen Screen-luokkaolion taulukon vapaavarastoon:

```
Screen *ps = new Screen[10];
```

Koska Sreen-luokalla on new[]()-jäsenoperaattori, new-lauseke kutsuu tuota operaattoria. Operaattorin size_t-parametri alustetaan automaattisesti arvolla, joka edustaa vaadittavan muistin kokoa tavuina kymmenen Screen-olion taulukon varaamiseen muistista.

Vaikka Screen-luokalla on new[]()-jäsenoperaattori, voi ohjelmoija käynnistää globaalin new[]()-operaattorin ja luoda Screen-oliotaulukon käyttämällä globaalin viittausalueen erotteluoperaattoria. Esimerkiksi seuraava new-lauseke käynnistää globaalilla viittausalueella määritellyn new[]()-operaattorin:

```
Screen *ps = ::new Screen[10];
```

delete[]()-jäsenoperaattorin paluutyypin pitää olla void ja ensimmäisen parametrin tyyppiä void*. Tässä on esimerkiksi Screen-luokkamme delete[]()-operaattorin esittely:

```
class Screen {
public:
    void operator delete[]( void* );
};
```

Luokkataulukon tuhoamisessa pitää delete-lausekkeessa käyttää seuraavaa syntaksia:

```
delete[] ps;
```

Kun tuollaisen delete-lausekkeen operandi on osoitin luokkatyyppiin, kääntäjä etsii luokasta delete[]()-jäsenoperaattoria. Jos se löytyy, valitaan tuo operaattori vapauttamaan taulukon muistialue; muussa tapauksessa kutsutaan globaalia delete[]()-operaattoria. Operaattorin void*-parametri alustetaan automaattisesti arvolla, joka edustaa muistialueen alkua, johon taulukko on tallennettu.

Vaikka Screen-luokalla on delete[]()-jäsenoperaattori, voi ohjelmoija käynnistää globaalin delete[]()-operaattorin valikoivasti käyttämällä globaalin viittausalueen erotteluoperaattoria. Esimerkiksi

```
::delete[] ps;
```

käynnistää globaalilla viittausalueella määritellyn delete[]()-operaattorin.

new[]()- tai delete[]()-luokkaoperaattorin lisäys tai poistaminen ei vaadi muutoksia käyttäjän koodiin. new- ja delete-lausekkeilla on sama muoto, kutsuttiinpa sitten globaalia tai luokan operaattoria.

new-lauseke, joka luo taulukon, kutsuu ensiksi luokan new[]()-operaattoria muistin varaamiseen ja sitten oletusmuodostajaa iteratiivisesti taulukon jokaiselle elementille alustamista varten. Jos luokassa on muodostaja, mutta ei oletusmuodostajaa, new-lauseke on virheellinen. Kun new-lausekkeella luodaan taulukoita, ei löydy syntaksia, jolla voitaisiin määrittää alkuarvoja taulukon elementeille tai argumentteja luokan muodostajalle.

delete-lauseke, joka poistaa taulukon, kutsuu ensin luokan tuhoajaa taulukon jokaisen elementin poistamiseen ja sitten luokan delete[]()-operaattoria muistin vapauttamiseen. On tärkeää, että delete-lausekkeessa käytetään taulukkosyntaksia taulukoiden yhteydessä. Jos seuraavassa lauseessa

```
delete ps;
```

ps viittaa luokkaolioiden taulukkoon, saa hakasulkujen ([]) puuttuminen aikaan sen, että tuhoaja käynnistetään vain taulukon ensimmäiselle elementille, vaikka tarkoitus olisi vapauttaa taulukon koko muistialue.

Luokan delete[]()-operaattorilla voi olla myös yhden sijasta kaksi parametria, joista toinen on esimääritelty size_t-tyyppi. Esimerkiksi:

```
class Screen {
public:
    // korvaa:
    // void operator delete( void* );
    void operator delete[]( void*, size_t );
};
```

Jos toinen parametri annetaan, kääntäjä alustaa sen automaattisesti tavukoolla, joka tarvitaan taulukon varaamiseen.

15.8.2 Asemointioperaattorit new()- ja delete()

Luokan new()-jäsenoperaattoria voidaan ylikuormittaa edellyttäen, että jokaisessa esittelyssä on yksilöllinen parametriluettelo. Luokan kaikkien new()-operaattorien ensimmäisen parametrin pitää olla size_t-tyyppinen. Esimerkiksi:

```
class Screen {
public:
    void *operator new( size_t );
    void *operator new( size_t, Screen* );
    // ...
};
```

Lisäparametrit alustetaan new-lausekkeessa määritetyillä *asemointiargumenteilla* (*placement arguments*). Esimerkiksi:

```
void func( Screen *start ) {
    Screen *ps = new (start) Screen;
    // ...
}
```

new-lausekkeen osa, joka tulee new-avainsanan jälkeen suluissa, edustaa asemointiargumentteja. Edellinen new-lauseke kutsuu new()-jäsenoperaattoria, joka saa kaksi parametria. Ensimmäinen parametri alustetaan automaattisesti arvolla, joka edustaa Screen-luokan kokoa tavuina. Toinen parametri alustetaan start-asemointiargumentin arvolla.

On myös mahdollista ylikuormittaa luokan delete()-jäsenoperaattoria. Sellaista operaattoria ei kuitenkaan koskaan käynnistetä delete-lausekkeesta. Ylikuormitettua delete()-operaattoria kutsuu implisiittisesti vain toteutus, jos new-lausekkeen kutsuma muodostaja (niin, tämä ei ole kirjoitusvirhe, me todella tarkoitamme new-lauseketta) heittää poikkeuksen. Tutkikaamme tarkemmin, milloin sellaista delete()-operaattoria käytetään.

Seuraavan new-lausekkeen

```
Screen *ps = new ( start ) Screen;
```

toimenpiteet ovat seuraavat:

1. Kutsuu luokan new(size_t, Screen*)-operaattoria.
2. Kutsuu sitten Screen-luokan oletusmuodostajaa olion alustamiseksi.
3. Alustaa sitten ps:n Screen-olion osoitteella.

Olettakaamme, että luokan new(size_t, Screen*)-operaattori varaa muistia kutsumalla globaalia new()-operaattoria. Kuinka luokan suunnittelija voi varmistua, että tällä new()-operaattorilla varattu muisti vapautetaan oikein, jos 2-vaiheessa kutsuttu Screen-muodostaja heittää poikkeuksen? Kuinka luokan suunnittelija voi suojautua sellaiselta käyttäjän koodilta, joka voi

saada aikaan muistin häviämistä? Luokan suunnittelija voi tehdä ylikuormitetun delete()-operaattorin, jota kutsutaan tällaisessa tilanteessa (ja vain tässä tilanteessa).

Jos luokan suunnittelija tekee ylikuormitetun delete()-operaattorin, jonka parametrien tyypit vastaavat new()-operaattorin parametrityyppejä, kutsuu toteutus automaattisesti tätä delete()-operaattoria muistin vapauttamiseksi. Jos esimerkiksi seuraavassa new-asemointilausekkeessa

```
Screen *ps = new (start) Screen;
```

Screen-luokan oletusmuodostaja päättyy poikkeuksen heittämiseen, toteutus etsii delete()-operaattoria Screen-luokan viittausalueelta. Jotta delete()-operaattori otettaisiin huomioon, pitää sen parametrien tyyppien vastata kutsutun new()-operaattorin tyyppejä. Koska new()-operaattorin ensimmäinen parametri on aina size_t-tyyppinen ja delete()-operaattorin ensimmäinen parametri on aina void*-tyyppinen, ei funktioiden ensimmäistä parametria oteta mukaan tähän vertailuun. Toteutus etsii Screen-luokasta seuraavan muotoista delete()-operaattoria:

```
void operator delete( void*, Screen* );
```

Jos delete()-operaattori löytyy Screen-luokasta, kutsutaan sitä vapauttamaan muisti, jos new-lausekkeen kutsuma muodostaja heittää poikkeuksen. Jos tätä delete()-operaattoria ei löydy, silloin ei kutsuta mitään delete()-operaattoria.

Luokan suunnittelija voi sitten päättää, tehdäkö tiettyä new()-lauseketta vastaava delete()-operaattori, sen perusteella, varaako new()-operaattori muistia vai käyttääkö se juuri varattua muistia uudelleen. Jos se varaa muistia, tulisi silloin tehdä delete()-asemointioperaattori muistin vapauttamista varten sellaista tilannetta silmällä pitäen, jossa new-lausekkeesta kutsuttu muodostaja heittää poikkeuksen. Ellei new()-asemointioperaattori varaa muistia, ei silloin tarvitse tehdä vastaavaa delete()-operaattoria muistin vapauttamiseen.

On myös mahdollista ylikuormittaa luokan new[]()- ja delete[]()-asemointioperaattoreita taulukoille:

```
class Screen {
public:
    void *operator new[]( size_t );
    void *operator new[]( size_t, Screen* );
    void operator delete[]( void*, size_t );
    void operator delete[]( void*, Screen* );
    // ...
};
```

new[]()-asemointioperaattoria käytetään, kun taulukkoa varaavalle new-lausekkeelle määritetään vastaavat asemointiargumentit. Esimerkiksi

```
void func( Screen *start ) {
    // kutsuu Screen::operator new[]( size_t, Screen* )
    Screen *ps = new (start) Screen[10];
    // ...
}
```

Jos tämän new-lausekkeen kutsuma muodostaja heittää poikkeuksen, toteutus kutsuu au-

tomaattisesti Screen-luokkaan määriteltyä vastaavaa ylikuormitettua delete[]()-operaattoria.

Harjoitus 15.9

Kerro, mitkä seuraavista alustuksista ovat virheellisiä, vai onko yksikään. Perustele, miksi.

```
class iStack {
public:
    iStack( int capacity )
        : _stack( capacity ), _top( 0 ) {}
    // ...
private:
    int _top;
    vector< int > _stack;
};

(a) iStack *ps = new iStack(20);
(b) iStack *ps2 = new const iStack(15);
(c) iStack *ps3 = new iStack[ 100 ];
```

Harjoitus 15.10

Kerro, mitä tapahtuu seuraavissa new- ja delete-lausekkeissa:

```
class Exercise {
public:
    Exercise();
    ~Exercise();
};

Exercise *pe = new Exercise[20];
delete[] pe;
```

Muuta new- ja delete-lausekkeita niin, että ne kutsuvat globaaleja new()- ja delete()-operaattoreita.

Harjoitus 15.11

Kerro, miksi luokan suunnittelijan tulisi tehdä luokalle delete()-asemointioperaattori?

15.9 Käyttäjän määrittelemät konversiot

Olemme jo nähneet, kuinka konversioita käytetään sisäisten tyyppien operandeille. Kohdassa 4.14 tutkimme, kuinka tyyppikonversioita käytetään sisäisten tyyppien operaattoreille. Kohdassa 9.3 tutkimme, kuinka tyyppikonversioita käytetään funktion kutsun argumenteille, jotta ne saataisiin funktion parametrien tyyppisiksi. Tyyppikonversioita käytetään esimerkiksi seuraavien kuuden lisäysoperaation operandeille:

```
char ch; short sh; int ival;
```

```
/* yksi operandi jokaisessa lisäyksessä
 * vaatii tyyppikonversion */

ch + ival;      ival + ch;
ch + sh;        sh + ch;
ival + sh;      sh + ival;
```

ch- ja sh-operandit ylennetään int-tyyppisiksi. Tapahtuva operaatio on kahden int-tyypin arvojen lisäys. Kääntäjä tekee ylennykset implisiittisesti eivätkä ne siten näy käyttäjälle.

Tässä kohtaa mietimme, kuinka luokan suunnittelija voi tehdä käyttäjän määrittelemiä konversioita luokkatyyppisille olioille. Myös kääntäjä käynnistää implisiittisesti näitä käyttäjän määrittelemiä konversioita tarpeen mukaan. Jotta voisimme kuvata, miksi käyttäjän määrittelemiä konversioita tarvitaan, palaamme SmallInt-luokkaamme, joka esiteltiin kohdassa 10.9.

Muista, että tämä luokka mahdollistaa määritellä olioita, jotka voivat sisältää samanlaisia arvoalueita kuin 8-bittinen unsigned char — tarkoittaa väliltä 0 — 255. Lisäksi se sieppaa ali- ja ylivuotovirheet. Sen lisäksi se käyttäytyy samalla tavalla kuin unsigned char.

Haluamme kyetä lisäämään ja vähentämään SmallInt-olioita sekä muista SmallInt-olioista että sisäisistä aritmeettisista tyypeistä. Voimme toteuttaa tämän tuen tekemällä kuusi SmallInt-operaattorifunktiota:

```
class SmallInt {
    friend operator+( const SmallInt , int );
    friend operator-( const SmallInt &, int );
    friend operator-( int, const SmallInt & );
    friend operator+( int, const SmallInt & );
public:
    SmallInt( int ival ) : value( ival ) { }
    operator+( const SmallInt & );
    operator-( const SmallInt & );
    // ...
private:
    int value;
};
```

Jäsenoperaattorit mahdollistavat sen, että voimme lisätä ja vähentää SmallInt-olioita muista SmallInt-olioista. Globaalit ystäväoperaattorit mahdollistavat, että voimme lisätä ja vähentää SmallInt-olioita sisäisistä aritmeettisista tyypeistä. Vain kuusi operaattoria on tarpeen, koska kaikki sisäiset aritmeettiset tyypit voidaan konvertoida vastaavaksi int-tyyppiseksi parametriksi. Esimerkiksi lauseke

```
SmallInt si( 3 );
si + 3.14159
```

ratkaistaan kahdessa vaiheessa:

1. double, joka on literaalivakio 3.14159, konvertoidaan kokonaislukuarvoksi 3.

2. Käynnistetään operator+(const SmallInt &,int)-operaattori, joka palauttaa arvon 6.

Jos haluamme tukea myös biteittäistä, loogista, vertailu- ja yhdistettyä sijoitusoperaattoria, vaadittavien operaattoreiden määrästä tulee — niin, uuvuttava. Sen sijaan, että tekisimme kaikki ylikuormitetut operaattorit, pitäisimme parempana tapaa, jossa konvertoisimme automaattisesti SmallInt-luokkaoliot int-tyyppisiksi olioiksi.

C++:ssa on mekanismi, jolla jokainen luokka voi määritellä konversiot, joita voidaan käyttää luokkatyyppinsä olioihin. SmallInt-luokalle määrittelemme konversion SmallInt-oliosta int-tyypiksi. Tässä on toteutus:

```
class SmallInt {
public:
    SmallInt( int ival ) : value( ival ) { }

    // konversio-operaattori:
    // SmallInt ==> int
    operator int() { return value; }

    // ylikuormitettuja operaattoreita ei tarvita

private:
    int value;
};
```

int()-operaattori on *konversiofunktio*, joka määrittelee *käyttäjän määrittelemän konversion*. Käyttäjän määrittelemä konversio tapahtuu luokkatyypin ja konversiofunktiossa mainitun tyypin välillä; tässä tapauksessa int-tyyppi. Konversiofunktion määrittely ilmaisee, mitä konversio tarkoittaa ja toimenpiteet, jotka kääntäjän pitää suorittaa, kun konversiota käytetään. Konversion SmallInt-oliosta int-tyypiksi tarkoitus on palauttaa int-tyypin arvo, joka on tallennettu value-tietojäseneen.

SmallInt-luokkaoliota voidaan nyt käyttää missä tahansa missä int-oliotakin. Olettaen, että ylikuormitettuja operaattoreita ei tehdä ja että SmallInt-luokkaan tehdään sen sijaan konversiofunktio int-tyypiksi, ratkaistaan seuraava yhteenlasku

```
SmallInt si( 3 );
si + 3.14159;
```

seuraavassa kahdessa vaiheessa:

1. SmallInt-luokan konversiofunktio käynnistetään, joka johtaa kokonaislukuarvoon 3.
2. Kokonaislukuarvo 3 ylennetään arvoon 3.0 ja lisätään double-tyyppiseen literaalivakioon 3.14159, joka puolestaan johtaa double-arvoon 6.14159.

Tämä piirre jäljittelee enemmän sisäisten tyyppien operandien käyttäytymistä verrattuna aikaisemmin määriteltyihin ylikuormitettuihin operaattoreihin. Kun int-tyyppinen arvo lisätään double-tyyppiseen arvoon, suoritettu operaatio on double-tyyppisten operandien yhteenlasku (int-tyyppinen arvo konvertoidaan double-tyyppiseksi) ja yhteenlaskun tulos johtaa double-tyyppiseen

arvoon.

Seuraava ohjelma kuvaa SmallInt-luokan käyttöä:

```
#include <iostream>
#include "SmallInt.h"
SmallInt si1, si2;

int main() {
    cout << "enter a SmallInt, please:  ";
    while ( cin >> si1 ) {
        cout << "The value read is "
            << si1 << "\nIt is ";

        // SmallInt::operator int() käynnistetty kahdesti
        cout << ( ( si1 > 127 )
              ? "greater than "
              : ( ( si1 < 127 )
                ? "less than "
                : "equal to ") ) << "127\n";

        cout << "\nenter a SmallInt, please \
              (ctrl-d to exit):  ";
    }
    cout << "bye now\n";
}
```

Kun ohjelma käännetään ja suoritetaan, se generoi seuraavat tulokset:

```
enter a SmallInt, please:  127

The value read is 127
It is equal to 127

enter a SmallInt, please (ctrl-d to exit):  126

The value read is 126
It is less than 127

enter a SmallInt, please (ctrl-d to exit):  128

The value read is 128
It is greater than 127

enter a SmallInt, please (ctrl-d to exit):  256
***SmallInt range error: 256 ***
```

SmallInt-luokan toteuttavaan koodiin on lisätty tukea, ja se näyttää nyt seuraavalta:

```
#include <iostream>

class SmallInt {
    friend istream&
        operator>>( istream &is, SmallInt &s );
    friend ostream&
        operator<<( ostream &os, const SmallInt &s )
        { return os << s.value; }
public:
    SmallInt( int i=0 ) : value( rangeCheck( i ) ){}
    int operator=( int i )
        { return( value = rangeCheck( i ) ); }
    operator int() { return value; }
private:
    int rangeCheck( int );
    int value;
};
```

Luokan rungon ulkopuolelle määritellyt jäsenfunktiot ovat seuraavassa:

```
istream& operator>>( istream &is, SmallInt &si ) {
    int ix;
    is >> ix;
    si = ix; // SmallInt::operator=(int)
    return is;
}

int SmallInt::rangeCheck( int i )
{
/* jos yksikin muu kuin 8 ensimmäistä bittiä on asetettu
 * on arvo liian suuri: raportoi ja poistu sitten */

    if ( i & ~0377 ) {
        cerr << "\en***SmallInt range error: "
            << i << " ***" << endl;
        exit( -1 );
    }
    return i;
}
```

15.9.1 Konversiofunktiot

Konversiofunktio on luokan erityinen jäsenfunktio. Se määrittelee käyttäjän määrittelemän konversion, jossa luokan olio konvertoidaan joksikin toiseksi tyypiksi. Konversiofunktio esitellään luokan rungossa määrittämällä avainsana operator ja sen jälkeen tyyppi, joka on konversion kohde.

Nimen, joka tulee konversiofunktion esittelyn operator-avainsanan jälkeen, ei tarvitse olla sisäinen tyypin nimi. Token-luokassa, joka on määritely seuraavassa, määritellään useita konversiofunktioita. Yksi konversiofunktio on määritelty käyttäen typedef-nimeä tName. Toinen konversiofunktio määrittelee konversion SmallInt-luokkatyyppiseksi:

```
#include "SmallInt.h"

typedef char *tName;
class Token {
public:
    Token( char*, int );
    operator SmallInt() { return val; }
    operator tName()    { return name; }
    operator int()      { return val; }
    // muut julkiset jäsenet
private:
    SmallInt val;
    char *name;
};
```

Huomaa, konversiofunktioiden määrittelyt SmallInt- ja int-tyypiksi ovat samanlaisia. Konversiofunktio Token::operator int() palauttaa val-jäsenen arvon. Koska val:in tyyppi on SmallInt, käytetään konversiofunktiota SmallInt::operator int() konvertoimaan val implisiittisesti int-tyypiksi. Kääntäjä käyttää itse Token::operator int() -konversiofunktiota implisiittisesti ja konvertoi Token-tyyppisen olion int-tyyppiseksi. Kääntäjä käyttää tätä konversiota implisiittisesti esimerkiksi konvertoidakseen Token-tyyppiset argumentit t1 ja t2 int-tyyppisiksi, joka on print():in parametrin tyyppi:

```
#include "Token.h"

void print( int i )
{
    cout << "print( int ) : " << i << endl;
}

Token t1( "integer constant", 127 );
Token t2( "friend", 255 );

int main()
{
    print( t1 ); // t1.operator int()
    print( t2 ); // t2.operator int()
```

```
    return 0;
}
```

Kun tämä pieni ohjelma käännetään ja suoritetaan, se generoi seuraavan tulostuksen:

```
print( int ) : 127
print( int ) : 255
```

Konversiofunktiolla on yleinen muoto

```
operator type();
```

jossa type korvataan sisäisellä tyypillä, luokkatyypillä tai typedef-nimellä. Konversiofunktioita, joissa type edustaa joko taulukkoa tai funktiotyyppiä, ei sallita. Konversiofunktion pitää olla jäsenfunktio. Sen esittelyssä ei saa määrittää paluutyyppiä eikä parametriluetteloa. Esimerkiksi seuraavat esittelyt ovat kaikki virheellisiä:

```
operator int( SmallInt & );  // virhe: ei ole jäsen

class SmallInt {
public:
    int operator int();      // virhe: paluutyyppi
    operator int( int = 0 ); // virhe: parametriluettelo
    // ...
};
```

Eksplisiittinen tyyppimuunnos voi aiheuttaa konversiofunktion käynnistymisen. Jos konvertoitava arvo on luokkatyyppinen, sillä on konversiofunktio ja sen tyyppiä käytetään tyyppimuunnoksessa, silloin kutsutaan konversiofunktiota. Esimerkiksi:

```
#include "Token.h"
Token tok( "function", 78 );

// toiminnallinen huomio: käynnistää Token::operator SmallInt()
SmallInt tokVal = SmallInt( tok );

// static_cast: käynnistää Token::operator tName()
char *tokName = static_cast< char * >( tok );
```

Konversiofunktiolla Token::operator char*() voi olla epätoivottu sivuvaikutus. Huomaatko, mikä se on? Kun yritetään käsitellä yksityistä jäsentä Token::name suoraan, se saa aikaan käännösvirheen:

```
char *tokName = tok.name; // virhe: Token::name on yksityinen
```

Kuitenkin konversiofunktioomme on juuri sellainen pääsy, jota etsimme suojellaksemme Token::name:n muuttamisen suoraan käyttäjien toimesta. Se ei ehkä ole sitä, mitä haluamme. Tässä on esimerkki siitä, kuinka sellainen muokkaus voi tapahtua:

```
#include "Token.h"
Token tok( "function", 78 );

char *tokName = tok; // ok: implisiittinen konversio
*tokName = 'P'; // hups: sanasen (token) nimi on nyt Punction!
```

Aikomuksemme on sallia vain lukuoikeus konvertoidulle Token-luokan oliolle. Jotta voisimme pakottaa tähän, pitää konversion operaattorin palauttaa const char* -tyyppi:

```
typedef const char *cchar;
class Token {
public:
    operator cchar() { return name; }
    // ...
};

// virhe: konversio tyypistä char* tyyppiin const char* ei ole sallittu
char *pn = tok;
const char *pn2 = tok; // ok
```

Toinen ratkaisu on muuttaa Token-luokan määrittelyä niin, että se käyttää C++-vakiokirjastossa määriteltyä string-tyyppiä. Esimerkiksi:

```
class Token {
public:
    Token( string, int );
    operator SmallInt() { return val; }
    operator string()   { return name; }
    operator int()      { return val; }
    // muut julkiset jäsenet
private:
    SmallInt val;
    string name;
};
```

Konversiofunktion Token::operator string() ajatus on palauttaa merkkijono, joka edustaa sanasen nimeä arvonaan. Tämä estää ohjelmaa erehdyksessä muokkaamasta Token-luokan yksityisen name-jäsenen arvoa.

Kun konversiofunktiota käytetään, pitääkö konversion kohdetyypin täsmätä konversiofunktion tyypin kanssa täsmälleen? Käynnistääkö esimerkiksi seuraava koodi Token-luokkaan määritellyn operator int() -konversiofunktion?

```
extern void calc( double );
Token tok( "constant", 44 );

// onko tok.operator int() käynnistetty? on.
```

```
// int --> double vakiokonversiolla
calc( tok );
```

Jos konversion kohde (tässä tapauksessa double-tyyppi) ei täsmää konversiofunktion tyypin kanssa (tässä tapauksessa int-tyyppi) täsmälleen, voidaan konversiofunktio silti käynnistää, jos kohdetyyppi voidaan saavuttaa vakiokonversion avulla. (Vakiokonversiot on kuvattu kohdassa 9.3). Jotta calc()-funktiota voitaisiin kutsua, käynnistetään Token::operator int(), joka konvertoi tok:n Token-tyypistä int-tyypiksi. Sitten voidaan käyttää vakiokonversiota käyttäjän määrittelemän konversion tulokseen int-tyypistä double-tyypiksi.

Vain vakiokonversiot ovat sallittuja käyttäjän määrittelemän konversion jälkeen. Jos kohdetyypin saavuttamiseksi on käytettävä toista käyttäjän määrittelemää konversiota, ei kääntäjä käytä konversiota implisiittisesti. Jos esimerkiksi Token ei määrittelisi operator int() -operaattoria, olisi seuraava kutsu kelpaamaton:

```
extern void calc( int );
Token tok( "pointer", 37 );

// ellei Token::operator int() olisi määritelty,
// saisi tämä kutsu aikaan käännösvirheen
calc( tok );
```

Ellei Token::operator int() ole määritelty, konversio tok-tyypistä int-tyyppiin vaatisi kaksi käyttäjän määrittelemää konversiofunktiota. tok-argumentti pitäisi ensin konvertoida Token-tyypistä SmallInt-tyyppiseksi käyttäen konversiofunktiota

```
Token::operator SmallInt()
```

ja tämän käyttäjän määrittelemän konversion tulos pitäisi sitten konvertoida int-tyyppiseksi käyttämällä konversiofunktiota

```
SmallInt::operator int()
```

Jos Token::operator int() on määrittelemättä, saa calc(tok)-kutsu aikaan käännösvirheen, koska implisiittistä konversiota Token-tyypistä int-tyypiksi ei ole olemassa.

Jos konversiofunktion tyypin ja luokkatyypin välillä ei ole mitään loogista yhteyttä, voi konversiofunktion teosta tulla moniselitteinen ohjelman lukijalle. Esimerkiksi:

```
class Date {
public:
    // arvaa, mikä jäsen palautetaan!
    operator int();
private:
    int month, day, year;
};
```

Minkä arvon tulisi Date-luokan operator int() -konversiofunktion palauttaa? Mikä valinta tehdäänkin ja mistä syystä, on Date-olioiden käyttö moniselitteistä ohjelman lukijalle, koska ei ole olemassa loogista yksi yhteen -yhteyttä Date-tyyppisen olion ja int-tyyppisen arvon välillä.

15.9.2 Muodostaja konversiofunktiona

Luokkien muodostajat, jotka saavat yhden parametrin kuten SmallInt-luokan SmallInt(int)-muodostaja, määrittelevät implisiittisten konversioiden joukon muodostajien parametrityypeistä SmallInt-tyypeiksi. Esimerkiksi SmallInt(int) konvertoi int-tyyppisiä arvoja SmallInt-arvoiksi.

```
extern void calc( SmallInt );
int i;

// pitää konvertoida i SmallInt-arvoksi
// SmallInt(int) toteuttaa tämän
calc( i );
```

i konvertoidaan calc(i)-kutsussa SmallInt-arvoksi käynnistämällä SmallInt(int)-muodostaja. Kääntäjä kutsuu muodostajaa, joka luo SmallInt-tyyppisen tilapäisolion. Tämän olion kopio välitetään sitten calc()-funktiolle. On sama kuin edellinen funktiokutsu olisi kirjoitettu seuraavasti:

```
// C++-pseudokoodia:
// luodaan tilapäinen SmallInt-olio
{
    SmallInt temp = SmallInt( i );
    calc( temp );
}
```

Esimerkin aaltosulut ilmaisevat tilapäiseksi generoidun SmallInt-olion elinkaarta: tilapäinen olio tuhotaan funktion kutsulauseen lopussa.

Muodostajan parametrin tyyppi voi olla toisen luokan tyyppi. Esimerkiksi:

```
class Number {
public:
    // luo Number-arvo SmallInt-arvosta
    Number( const SmallInt & );
```

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

Missä tapauksessa SmallInt-tyypin arvoa voidaan käyttää siellä, missä Number-tyypin arvoa tarvitaan? Esimerkiksi:

```
extern void func( Number );
SmallInt si(87);

int main()
{   // käynnistä Number( const SmallInt & )
    func( si );
    // ...
}
```

Kun muodostajaa käytetään implisiittisen konversion suorittamiseen, pitääkö muodostajan parametrin tyypin vastata konvertoitavaa tyyppiä täsmälleen? Käynnistääkö esimerkiksi seuraava koodi `SmallInt(int)`-muodostajan, joka on määritelty SmallInt-luokkaan konvertoimaan `dobj`-olion SmallInt-tyyppiseksi?

```
extern void calc( SmallInt );
double dobj;

// onko SmallInt(int) käynnistetty? on.
// dobj konvertoitu double-tyypistä int-tyypiksi vakiokonversiolla
calc( dobj );
```

Jos on tarpeen, argumenttiin käytetään vakiokonversiota ennen kuin muodostajaa kutsutaan tekemään käyttäjän määrittelemä konversio. Jotta `calc()`-funktiota voitaisiin kutsua, käytetään vakiokonversiota `dobj`-olion konversioon `double`-tyypistä `int`-tyypiksi. Sitten käynnistetään `SmallInt(int)` konvertoimaan tulos, joka syntyi konversiosta SmallInt-tyypiksi.

Kääntäjä käyttää muodostajaa implisiittisesti yhdellä parametrilla ja konvertoi parametrien tyypit muodostajan luokan tyyppisiksi. Tämä ei aina ehkä ole se, mitä haluamme. Voimme päättää, että `Number(const SmallInt&)`-muodostajaa pitää käyttää vain alustamaan Number-tyyppistä oliota, jonka arvo on SmallInt-tyyppinen ja että kääntäjän ei tulisi muutoin koskaan käyttää tätä muodostajaa implisiittisiin tyyppikonversioihin. Jotta voisimme estää muodostajan käytön implisiittisesti tyyppikonversioihin, voimme esitellä muodostajan eksplisiittiseksi:

```
class Number {
public:
    // ei käytetä koskaan implisiittisiin konversioihin
    explicit Number( const SmallInt & );
    // ...
};
```

Kääntäjä ei koskaan käytä eksplisiittistä muodostajaa implisiittisten tyyppikonversioiden suorittamiseen. Esimerkiksi:

```
extern void func( Number );
SmallInt si(87);

int main()
{    // virhe: ei implisiittistä konversiota SmallInt-tyypistä Number-tyypiksi
     func( si );
     // ...
}
```

Muodostajaa voidaan kuitenkin käyttää tyyppikonversioiden suorittamiseen niin kauan, kuin ohjelma pyytää konversioita eksplisiittisesti tyyppimuunnoksen muodossa. Esimerkiksi:

```
SmallInt si(87);

int main()
{    // virhe: ei implisiittistä konversiota SmallInt-tyypistä Number-tyypiksi
     func( si );
     func( Number( si ) ); // ok: tyyppimuunnos
     func( static_cast< Number >( si ) ); // ok: tyyppimuunnos
}
```

15.10 Konversion valinta ◆

Käyttäjän määrittelemä konversio on sellainen, jonka suorittaa joko konversiofunktio tai muodostaja. Kuten olemme nähneet, konversiofunktion tekemän konversion jälkeen voidaan suorittaa vakiokonversio, jolla konversiofunktion tulos saadaan kohdetyypin mukaiseksi. Samalla tavalla muodostajan suorittamaa konversiota voi edeltää vakiokonversio, jolla arvo saadaan konvertoitua muodostajan parametrin tyyppiseksi.

Käyttäjän määrittelemä konversiosarja (*user-defined conversion sequence*) on yhdistelmä tarvittavista käyttäjän määrittelemistä konversioista ja vakiokonversioista, joilla arvo saadaan konversion kohdetyypiksi. Käyttäjän määrittelemän konversiosarjan muoto on seuraava:

```
Vakiokonversiosarja ->
    Käyttäjän määrittelemä konversio ->
        Vakiokonversiosarja
```

jossa käyttäjän määrittelemä konversio käynnistää joko funktion tai muodostajan.

Kun arvoa yritetään konvertoida, on mahdollista, että voidaan käyttää kahta käyttäjän määrittelemää konversiosarjaa arvon konvertoimiseen kohdetyypiksi. Kun useampaa kuin yhtä konversiosarjaa voidaan käyttää, kääntäjän pitää valita paras sarja konversion suorittamiseen. Tässä kohtaa katsomme, kuinka se tehdään.

Luokkaan voidaan määritellä useita konversiofunktioita, joita voidaan käyttää konversioon. Esimerkiksi Number-luokkaamme voidaan määritellä kaksi konversiofunktiota: operator

int() ja operator float(). Molempia funktioita voidaan käyttää, kun halutaan konvertoida Number-tyyppinen olio float-tyyppiseksi arvoksi. Tietysti Token::operator float() -konversiota voitaisiin käyttää suoraan float-tyypin muodostamiseksi. Token::operator int() -konversiota voitaisiin myös käyttää, koska on mahdollista saada konversiofunktion tulos, int, vakiokonversion kautta float-tyypiksi. Onko konversio moniselitteinen, koska on olemassa kaksi käyttäjän määrittelemää konversiosarjaa? Vai onko toinen käyttäjän määrittelemistä konversiosarjoista parempi?

```
class Number {
public:
    operator float();
    operator int();
    // ...
};
Number num;
float ff = num; // mikä konversiofunktio? operator float()
```

Jos voidaan käyttää kahta konversiofunktiota, käytetään sitä vakiokonversiota parhaimman käyttäjän määrittelemän konversiosarjan valinnan apuna, joka seuraa konversiofunktiota. Edellisessä esimerkissä voidaan käyttää kahta seuraavaa käyttäjän määrittelemää konversiosarjaa konversion tekemiseen:

1. operator float() -> vastaa täysin
2. operator int() -> vakiokonversio

Kuten näimme kohdassa 9.3, täysi vastaavuus on parempi konversio kuin vakiokonversio. Ensimmäinen konversiosarja on siten parempi ja Token::operator float() -konversiofunktio tulee valituksi konversion tekemiseen.

Samalla tavalla on mahdollista, että kahta muodostajaa voidaan käyttää arvon konvertoimiseen kohdetyypiksi. Tässä tapauksessa käytetään vakiokonversiota, joka edeltää käyttäjän määrittelemää konversiota, parhaan käyttäjän määrittelemän konversiosarjan valintaan. Esimerkiksi:

```
class SmallInt {
public:
    SmallInt( int ival ) : value( ival ) { }
    SmallInt( double dval )
        : value( static_cast< int >( dval ) )
    { }
};

extern void manip( const SmallInt & );

int main() {
    double dobj;
    manip( dobj ); //ok: SmallInt( double )
}
```

Tässä SmallInt-luokassa määritellään kaksi muodostajaa — SmallInt(int) ja SmallInt(double) — joita voidaan käyttää konversioon double-tyyppisestä dobj-oliosta SmallInt-tyyppiseksi arvoksi. SmallInt(double)-muodostajaa voidaan käyttää konversioon, koska se saa parametrin suoraan double-tyyppisenä. SmallInt(int)-muodostajaa voidaan myös käyttää, koska on mahdollista konvertoida double-tyyppinen dobj-olio int-tyyppikseksi vakiokonversiolla ja käyttää tätä tulosta SmallInt(int)-muodostajan argumenttina. Siten voidaan käyttää seuraavia kahta käyttäjän määrittelemää konversiosarjaa:

1. vastaa täysin -> SmallInt(double)
2. vakiokonversio -> SmallInt(int)

Koska täysi vastaavuus on parempi kuin vakiokonversio, käytetään SmallInt(double)-muodostajaa konversioon.

Aina ei ole mahdollista valita yhtä käyttäjän määrittelemää konversiosarjaa konversion suorittamiseen. Kaikki konversiosarjat voivat olla yhtä hyviä, jolloin sanomme, että konversio on *moniselitteinen*. Siinä tapauksessa kääntäjä ei käytä implisiittistä konversiota. Esimerkiksi, kun Number-luokkaan on määritelty kaksi konversiofunktiota:

```
class Number {
public:
    operator float();
    operator int();
    // ...
};
```

ei ole mahdollista konvertoida implisiittisesti Number-tyyppistä oliota long-tyyppiseksi olioksi. Seuraava lause saa aikaan käännösvirheen, koska käyttäjän määrittelemät konversiosarjat ovat moniselitteisiä:

```
// virhe: molempia operaattoreita, float() ja int(), voidaan käyttää
long lval = num;
```

Käyttäjän määrittelemät konversiosarjat, joita voitaisiin käyttää num:in konvertoimiseen long-tyyppiseksi arvoksi, ovat seuraavat:

1. operator float() -> vakiokonversio
2. operator int() -> vakiokonversio

Koska käyttäjän määrittelemien konversioiden jälkeiset molemmat konversiot ovat vakiokonversioita, ne ovat yhtä hyviä eikä kääntäjä voi valita yksilöllistä konversiosarjaa implisiittiselle konversiolle.

Ohjelmoija voi ilmaista käytettävän konversiosarjan määrittämällä eksplisiittisen tyyppikonversion. Esimerkiksi:

```
// ok: eksplisiittinen tyyppikonversio
long lval = static_cast< int >( num );
```

Konversio on sitten eksplisiittinen, ja siinä käytetään Token::operator int() -konversiofunktiota ja sen jälkeen vakiokonversiota int-tyypistä long-tyypiksi.

Kun käyttäjän määrittelemää konversiota valitaan implisiittistä konversiota varten, voi myös syntyä moniselitteisyyttä, kun kaksi luokkaa määrittelevät konversioita toisilleen. Esimerkiksi:

```
class SmallInt {
public:
    SmallInt( const Number & );
    // ...
};

class Number {
public:
    operator SmallInt();
    // ...
};

extern void compute( SmallInt );
extern Number num;

compute( num ); // virhe: kaksi konversiota mahdollisia
```

num-argumentti voidaan konvertoida SmallInt-tyyppiseksi kahdella eri tavalla. Voidaan käyttää SmallInt::SmallInt(const Number&)-muodostajaa tai Number::operator SmallInt() -konversiofunktiota. Koska nämä kaksi funktiota ovat yhtä hyviä, kutsu on virheellinen.

Ohjelmoija voi kutsua Number-luokan konversiofunktiota eksplisiittisesti ja ratkaista moniselitteisyyden:

```
// ok: eksplisiittinen käynnistys ratkaisee moniselitteisyyden
compute( num.operator SmallInt() );
```

Ohjelmoija ei voi kuitenkaan ratkaista moniselitteisyyttä eksplisiittistä tyyppimuunnosta käyttäen, koska sekä konversiofunktiota että muodostajaa pidetään konversioina, jotka määritetään eksplisiittisesti tyyppimuunnoksena:

```
compute( SmallInt( num ) ); // virhe: yhä moniselitteinen
```

Kuten näet, tekemällä monia konversiofunktioita ja muodostajia implisiittisiä konversioita varten voi johtaa yllättäviin tuloksiin. Konversioita ja muodostajia tulisi käyttää järkevästi. On mahdollista rajoittaa muodostajien käyttöä implisiittisiin konversioihin (ja siten rajoittaa niiden yllättäviä sivuvaikutuksia) esittelemällä muodostajat eksplisiittisiksi.

15.10.1 Funktion ylikuormituksen ratkaisu — hieman lisää

Luvussa 9 kuvataan yksityiskohtaisesti, kuinka kutsu ylikuormitettuun funktioon ratkaistaan. Kun funktion kutsun argumentit ovat luokkatyyppisiä, osoittimia luokkatyyppeihin tai osoittimia jäseniin, otetaan suuri joukko funktioita huomioon funktioehdokkaina kutsua varten. Siten luokkatyyppisten argumenttien käyttö vaikuttaa funktion ylikuormituksen ratkaisun ensimmäiseen vaiheeseen — funktioehdokkaitten valintaan.

Funktion ylikuormituksen ratkaisun ensimmäisen vaiheen aikana valitaan parhaiten täsmäävä funktio. Jotta valinta voitaisiin tehdä, tyyppikonversiot, jotka konvertoivat argumentit vastaavien parametrien tyyppisiksi, laitetaan paremmuusjärjestykseen. Luokkatyyppisille argumenteille ja parametreille mahdollisiin konversioihin pitää ottaa mukaan myös edellisessä kohdassa esitellyt käyttäjän määrittelemät konversiosarjat. Funktion ylikuormituksen ratkaisuprosessin kolmannen vaiheen pitää sitten laittaa paremmuusjärjestykseen käyttäjän määrittelemät konversiosarjat.

Tässä kohtaa tutkimme tarkemmin, kuinka luokkatyyppiset argumentit ja parametrit vaikuttavat funktioehdokkaisiin ja kuinka käyttäjän määrittelemät konversiot vaikuttavat parhaiten täsmäävän funktion valintaan.

15.10.2 Funktioehdokkaat

Funktioehdokas on sellainen, jolla on sama nimi kuin kutsutulla funktiolla. Sanokaamme esimerkiksi, että meillä on seuraavanlainen funktiokutsu:

```
SmallInt si(15);
add( si, 566 );
```

Kutsun funktioehdokkaan nimen pitää olla add. Mitkä add()-funktion esittelyt otetaan huomioon?

Kuten kaikissa funktiokutsuissa, add()-funktion esittelyt, jotka ovat näkyvissä kutsupaikassa, ovat funktioehdokkaita. Esimerkiksi kaksi add()-funktiota, jotka on esitelty globaalilla viittausalueella, ovat seuraavan kutsun funktioehdokkaita:

```
const matrix& add( const matrix &, int );
double add( double, double );

int main() {
    SmallInt si(15);
    add( si, 566 );
    // ...
}
```

Kun mietitään funktioita, jotka ovat näkyvissä kutsupaikassa ja joiden argumentit ovat luokkatyyppisiä, ei niiden käsittelyssä ole eroa suhteessa muihin funktioihin. Kuitenkin sellaisten funktiokutsujen yhteydessä etsitään funktioiden esittelyitä myös kahdelta muulta viittausalueelta:

1. Jos argumentti on luokkatyyppinen olio, osoitin luokkatyyppiin, viittaus luokkatyyppiin tai osoitin luokan jäseneen ja jos luokkatyyppi on esitelty käyttäjän esittelemässä nimiavaruudessa, tuossa nimiavaruudessa esitellyt funktiot, joiden nimet ovat samoja kuin kutsutulla funktiolla, lisätään funktioehdokkaitten joukkoon. Esimerkiksi:

```
namespace NS {
    class SmallInt { /* ... */ };
    class String { /* ... */ };
    String add( const String &, const String & );
}

int main() {
    // si:n tyyppi on SmallInt-luokka:
    // luokka on esitelty NS-nimiavaruudessa
    NS::SmallInt si(15);

    add( si, 566 ); // NS::add() on funktioehdokas
    return 0;
}
```

   `si`-argumentin tyyppi on SmallInt, joka on NS-nimiavaruudessa esitelty luokkatyyppi. Funktio `add(const String&, const String&)`, joka on esitelty NS-nimiavaruudessa, lisätään funktioehdokkaitten joukkoon.

2. Jos argumentti on luokkatyyppinen olio, osoitin luokkatyyppiin, viittaus luokkatyyppiin tai osoitin luokan jäseneen ja luokalla on samannimisiä ystäväfunktioita kuin kutsuttu funktio, lisätään ystäväfunktiot funktioehdokkaitten joukkoon. Esimerkiksi:

```
namespace NS {
    class SmallInt {
        friend SmallInt add( SmallInt, int ) { /* ... */ }
    };
}

int main() {
    NS::SmallInt si(15);

    add( si, 566 ); // ystävä add() on funktioehdokas
    return 0;
}
```

   Funktioargumentin `si` tyyppi on SmallInt. Sen ystäväfunktio, `add(SmallInt, int)`, on NS-nimiavaruuden jäsen, vaikka sitä ei ole koskaan esitelty NS-nimiavaruudessa suoraan. Normaali etsintä NS-nimiavaruudesta ei löydä ystäväfunktiota. Kuitenkin `add()`-funktion kutsussa, jossa funktiolle annetaan SmallInt-tyyppinen argumentti, otetaan huomioon ystäväfunktiot, jotka on esitelty SmallInt-luokan jäsenluettelossa ja ne lisätään funktioehdokkaitten joukkoon.

Täten, jos funktion kutsussa on argumentti, joka on luokkatyyppinen olio, osoitin luokkatyyppiin, viittaus luokkatyyppiin tai osoitin luokan jäseneen, ovat funktioehdokkaat niiden funktioiden yhdiste, jotka ovat näkyvissä kutsupaikalla, jotka on esitelty nimiavaruudessa, jossa luokkatyyppi on määritelty ja jotka on esitelty ystävinä luokan jäsenluettelossa.

Jos laitamme kaikki nämä aikaisempien esimerkkien palaset yhteen

```
namespace NS {
    class SmallInt {
        friend SmallInt add( SmallInt, int ) { /* ... */ }
    };
    class String { /* ... */ };
    String add( const String &, const String & );
}

const matrix& add( const matrix &, int );
double add( double, double );

int main() {
    // si:n tyyppi on SmallInt-luokka:
    // luokka on esitelty NS-nimiavaruudessa
    NS::SmallInt si(15);

    add( si, 566 ); // kutsuu ystäväfunktiota
    return 0;
}
```

funktioehdokkaat ovat:

1. globaalit funktiot:

   ```
   add( const matrix &, int )
   add( double, double )
   ```

2. nimiavaruuden funktio:

   ```
   NS::add( const String &, const String & )
   ```

3. ystäväfunktio:

   ```
   NS::add( SmallInt, int )
   ```

Funktion ylikuormituksen ratkaisu valitsee SmallInt-luokan ystäväfunktion, `NS::add(SmallInt,int)`, parhaiten kutsua täsmäävänä, koska kutsussa määritetyt molemmat argumentit vastaavat ystäväfunktion parametreja täsmälleen.

Tietysti funktion kutsussa voi olla useampi kuin yksi luokkatyyppinen argumentti, osoitin luokkatyyppiin, viittaus luokkatyyppiin tai osoitin luokan jäseneen. Näitä argumentteja vastaavat luokkatyypit voivat olla erilaisia. Jokainen argumentti tutkitaan vuorollaan, jotta löydettäisiin funktioehdokkaat nimiavaruudesta, jossa luokka on määritelty ja luokan ystäväfunktioista. Tästä syystä kutsun, jonka argumentit ovat luokkatyyppisiä, funktioehdokkaisiin voi kuulua funktioita eri nimiavaruuksista ja ystäväfunktioita, jotka on esitelty eri luokissa.

15.10.3 Funktiokutsujen funktioehdokkaat luokan viittausalueella

Kun funktion kutsu, joka on muotoa

```
calc(t)
```

esiintyy luokan viittausalueella (esimerkiksi jäsenfunktiossa), voi ensimmäiseen funktioehdokkaitten joukkoon, jotka kuvattiin edellisessä alikohdassa (tarkoittaa niitä, joiden funktioesittelyt ovat näkyvissä kutsupaikassa), voi kuulua funktioita, jotka eivät ole jäsenfunktioita. Nimiresoluutiota käytetään niiden funktioehdokkaitten etsimiseen, jotka ovat näkyvissä kutsupaikassa. Nimiresoluutiota luokan viittausalueella on käsitelty laajasti kohdassa 13.9, sisäkkäisillä luokilla kohdassa 13.10, luokilla nimiavaruuden jäseninä kohdassa 13.11 ja paikallisilla luokilla kohdassa 13.12.

Tutkikaamme esimerkkiä:

```
namespace NS {
    struct myClass {
        void k( int );
        static void k( char* );
        void mf();
    };
    int k( double );
};

void h(char);

void NS::myClass::mf() {
    h('a'); // kutsuu globaalia h( char )-funktiota
    k(4);   // kutsuu myClass::k( int )-funktiota
}
```

Kuten mainittiin kohdassa 13.11, määreitä `NS::myClass::` etsitään käänteisessä järjestyksessä aloittaen ensin myClass-luokasta, sitten NS-nimiavaruudesta, jotta löydettäisiin näkyvä esittely nimelle, jota on käytetty `mf()`-jäsenfunktion määrittelyssä. Mietitäänpä ensin kutsua

```
h( 'a' );
```

Nimiresoluutio ottaa `h()`:n kutsussa ensin huomioon `mf()`-jäsenfunktion määrittelyn myClass-luokan jäsenfunktiot. Koska yhtään `h()`-nimistä jäsenfunktiota ei löydy myClass-luokan viittausalueelta, tutkitaan funktioehdokkaita sitten NS-nimiavaruudesta. Koska yhtään `h()`-nimistä funktiota ei löydy NS-nimiavaruudesta, etsitään funktioehdokkaita sitten globaalilta viittausalueelta. Globaali `h(char)`-funktio löytyy ja on ainoa funktioehdokkaitten joukossa, joka on näkyvissä kutsupaikassa.

Tämän etsinnän aikana, niin pian kuin funktion esittely löytyy, kutsupaikassa näkyvissä olevien funktioehdokkaitten etsintä päättyy. Tähän joukkoon kuuluvat vain ne funktiot, jotka on esitelty sillä viittausalueella, jossa nimiresoluutio onnistuu. Tämä voidaan nähdä funktioehdokkaitten joukosta kutsussa

```
k( 4 );
```

Luokan myClass viittausalue otetaan ensin huomioon kutsun funktioehdokkaille. Kaksi jäsenfunktiota, k(int) ja k(char*), löytyvät. Koska funktioehdokkaisiin, jotka ovat näkyvissä kutsupaikassa, kuuluvat vain ne funktiot, jotka on esitelty sillä viittausalueella, jossa nimiresoluutio onnistuu, ei NS-nimiavaruudesta etsitä funktioehdokkaita ja k(double)-funktio jätetään funktioehdokkaitten ulkopuolelle.

Jos ylikuormituksen ratkaisu huomaa kutsun olevan moniselitteinen, koska funktioehdokkaitten joukosta ei löydy sopivaa, on funktion kutsu virheellinen. Ympäröiviltä viittausalueilta ei etsitä lisää funktioehdokkaita, jotka paremmin vastaisivat funktion kutsun argumentteja.

15.10.4 Käyttäjän määrittelemät konversiosarjat paremmuusjärjestykseen

Funktion kutsun argumentti voidaan konvertoida funktioparametrin tyypiksi käyttämällä käyttäjän määrittelemää konversiosarjaa. Kuinka käyttäjän määrittelemät konversiosarjat vaikuttavat funktion ylikuormituksen ratkaisuun? Olkoon esimerkiksi seuraava kutsu calc()-funktioon. Mitä funktiota kutsutaan?

```
class SmallInt {
public
    SmallInt( int );
};

extern void calc( double );
extern void calc( SmallInt );
int ival;

int main() {
    calc( ival ); // mitä calc()-funktiota kutsutaan?
}
```

Funktio parametreineen, joka parhaiten vastaa funktion kutsun argumentteja, tulee valituksi. Tätä funktiota kutsutaan parhaiten täsmääväksi funktioksi tai parhaiten elinkelpoiseksi funktioksi. Jotta parhaiten elinkelpoinen funktio voidaan valita, laitetaan funktion argumenttien implisiittiset konversiot paremmuusjärjestykseen. Parhaiten elinkelpoinen funktio on sellainen, jonka argumentteihin käytetyt konversiot *eivät ole huonompia* kuin konversiot, jotka ovat tarpeellisia kutsuttaessa mitä tahansa muuta elinkelpoista funktiota, ja joidenkin argumenttien konversiot ovat *parempia* kuin konversiot, jotka ovat tarpeellisia samoille argumenteille, kun kutsutaan mitä tahansa muuta elinkelpoista funktiota.

Vakiokonversio on aina parempi kuin käyttäjän määrittelemä konversiosarja. Mitä tulee esimerkiksi edellisen esimerkin calc()-kutsuun, molemmat calc()-funktiot ovat elinkelpoisia funktioita. Funktio calc(double) on elinkelpoinen, koska on olemassa vakiokonversio, jolla int-tyyppinen argumentti voidaan konvertoida double-tyyppiseksi parametriksi. Funktio calc(SmallInt) on myös elinkelpoinen, koska on olemassa käyttäjän määrittelemä konversiosarja, joka konvertoi int-tyyppisen argumentin SmallInt-tyyppiseksi funktion parametriksi. Tämä käyttäjän määrittelemä konversio käyttää SmallInt(int)-muodostajaa konversion tekemiseen. Koska vakiokonversiosarja on

parempi kuin käyttäjän määrittelemä konversiosarja, valitaan parhaiten elinkelpoiseksi funktioksi calc(double) kutsua varten.

Mutta mitä jos vertaillaan kahta käyttäjän määrittelemää konversiosarjaa? Jos kaksi käyttäjän määrittelemää konversiosarjaa käyttävät eri konversiosarjoja tai eri muodostajia, pidetään molempia konversiosarjoja yhtä hyvinä. Esimerkiksi:

```
class Number {
public:
    operator SmallInt();
    operator int();
    // ...
};

extern void calc( int );
extern void calc( SmallInt );
extern Number num;

calc( num ); // virhe: moniselitteinen
```

Sekä calc(int) että calc(SmallInt) ovat elinkelpoisia funtkioita. Funktio calc(int) on elinkelpoinen, koska konversiofunktio Number::operator int() voi konvertoida Number-tyyppisen argumentin int-tyyppiseksi parametriksi. Funktio calc(SmallInt) on myös elinkelpoinen, koska konversiofunktio Number::operator SmallInt() voi konvertoida Number-tyyppisen argumentin SmallInt-tyyppiseksi funktion parametriksi. Koska käyttäjän määrittelemät konversiot ovat paremmuudessaan yhtä hyviä, kääntäjä ei voi päätellä, kumpi käyttäjän määrittelemistä konversiosarjoista on parempi. Edellinen funktion kutsu on siten moniselitteinen ja saa aikaan käännösvirheen.

Ohjelmoija voi ratkaista moniselitteisyyden tekemällä konversiosta eksplisiittisen. Esimerkiksi:

```
// eksplisiittinen konversio ratkaisee moniselitteisyyden
calc( static_cast< int >( num ) );
```

Eksplisiittinen tyyppimuunnos pakottaa kääntäjän konvertoimaan num-argumentin int-tyypiksi käyttäen konversiofunktiota Number::operator int(). Argumentin tyyppi on sitten int, joka täsmää calc(int)-funktioon ja valitaan parhaiten elinkelpoiseksi funktioksi.

Oletetaan, että Number-luokassa ei ole määritelty konversiofunktiota Number::operator int(). Olisiko kutsu

```
// Vain Number::operator SmallInt() määritelty
calc( num ); // yhä moniselitteinen?
```

yhä moniselitteinen? Muista, että SmallInt-luokassa on myös määritelty konversiofunktio, joka konvertoi SmallInt-tyyppisen arvon int-tyyppiseksi.

```
class SmallInt {
public:
    operator int();
    // ...
```

```
};
```

Voitaisiin olettaa, että calc(int)-funktiota voidaan silti kutsua konvertoimalla ensin num-argumentti Number-tyypistä SmallInt-tyyppiseksi käyttäen konversiofunktiota Number::operator SmallInt() ja konvertoida sitten tulos int-tyyppiseksi käyttäen konversiofunktiota SmallInt::operator int(). Näin ei kuitenkaan ole asianlaita. Muista, että vain käyttäjän määrittelemä konversio voi olla osa käyttäjän määrittelemästä konversiosarjasta. Ensimmäisen käyttäjän määrittelemän konversion jälkeen katsotaan tulevan vain vakiokonversioita. Ellei konversiota Number::operator int() ole määritelty, ei calc(int)-funktiota katsota elinkelpoiseksi funktioksi, koska ei ole olemassa implisiittistä konversiota, jolla voitaisiin konvertoida num-argumentti int-tyyppiseksi funktion parametriksi.

Täten, jos konversiota Number::operator int() ei ole määritelty, on calc(SmallInt) ainoa elinkelpoinen funktio. Se valittaisiin parhaiten elinkelpoiseksi funktioksi.

Jos kaksi käyttäjän määrittelemää konversiosarjaa käyttää samaa konversiofunktiota, käytetään parhaan käyttäjän määrittelemän konversiosarjan valitsemiseen konversiofunktion jälkeen suoritettavia vakiokonversioita laittamalla ne paremmuusjärjestykseen. Esimerkiksi:

```
class SmallInt {
public:
    operator int();
    // ...
};

void manip( int );
void manip( char );

SmallInt si ( 68 );

main() {
    manip( si ); // kutsuu funktiota manip( int )
}
```

Sekä manip(int) että manip(char) ovat elinkelpoisia funktioita. Funktio manip(int) on elinkelpoinen, koska konversiofunktio SmallInt::operator int() voi konvertoida SmallInt-tyyppisen argumentin int-tyyppiseksi parametriksi. Funktio manip(char) on myös elinkelpoinen, koska konversiofunktio SmallInt::operator int() voi konvertoida SmallInt-tyyppisen argumentin int-tyypiksi ja sitten vakiokonversio voi konvertoida tuloksen char-tyypiksi. Käyttäjän määrittelemät sarjat ovat siten

```
manip(int)  : operator int() -> täysi vastaavuus
manip(char): operator int() -> vakiokonversio
```

Koska molemmat käyttäjän määrittelemät konversiosarjat käyttävät samaa konversiofunktiota, käytetään vakiokonversioiden paremmuusjärjestystä parhaan sarjan päättelyyn. Koska täysi vastaavuus on parempi kuin vakiokonversio, valitaan manip(int)-funktio parhaiten elinkelpoiseksi funktioksi.

Huomaa, että vakiokonversiosarjoja, jotka tulevat käyttäjän määrittelemien konversiosar-

jojen jälkeen, käytetään vain valinnan kriteerinä, jos kaksi käyttäjän määrittelemää konversiosarjaa käyttävät samaa konversiofunktiota. Tämä poikkeaa hieman esimerkeistä, jotka esiteltiin kohdan 15.9 lopussa. Tässä kohtaa kuvaamme, kuinka kääntäjä valitsee käyttäjän määrittelemän konversion tietyn tyyppisen arvon konvertoimiseksi tietyn kohdetyypin mukaiseksi. Siinä tapauksessa sekä lähde- että kohdetyypit ovat kiinteitä ja kääntäjä valitsee niiden käyttäjän määrittelemien konversioiden väliltä, jotka tekevät konversioita näiden kahden tyypin välillä. Tässä otetaan huomioon kaksi erilaista funktiota, joilla on erilaiset parametrityypit ja kohdetyypit vaihtelevat. Jos kaksi eri parametrityyppiä vaativat erilaisia käyttäjän määrittelemiä konversiota, ei ole mahdollista valita, että yksi parametrityyppi olisi parempi kuin toinen, elleivät käyttäjän määrittelemät konversiot jaa samaa konversiofunktiota. Siinä tapauksessa voimme käyttää parhaan kohdetyypin valitsemiseen vakiokonversioita, jotka tulevat käyttäjän määrittelemien konversioiden jälkeen. Esimerkiksi:

```
class SmallInt {
public:
    operator int();
    operator float();
    // ...
};

void compute( float );
void compute( char );

SmallInt si ( 68 );

main() {
    compute( si ); // moniselitteinen
}
```

Sekä compute(float) että compute(char) ovat elinkelpoisia funktioita. Funktio compute(float) on elinkelpoinen, koska konversiofunktio SmallInt::operator float() voi konvertoida SmallInt-tyyppisen argumentin float-tyyppiseksi funktion parametriksi. Funktio compute(char) on myös elinkelpoinen, koska konversiofunktio SmallInt::- operator int() voi konvertoida SmallInt-tyyppisen argumentin int-tyypiksi ja sitten vakiokonversio voi konvertoida tuloksen char-tyypiksi. Käyttäjän määrittelemät sarjat ovat siten

compute(float): operator float() -> täysi vastaavuus
compute(char): operator int() -> vakiokonversio

Koska molemmat käyttäjän määrittelemät konversiosarjat käyttävät eri konversiofunktioita, ei ole mahdollista päätellä, millä funktiolla on paras parametri kutsua varten. Vakiokonversioiden paremmuusjärjestystä ei käytetä parhaan konversiosarjan ja siten parhaan parametrityypin päättelyyn. Kääntäjä ilmoittaa kutsun olevan moniselitteinen.

Harjoitus 15.12

C++-vakiokirjaston luokkiin ei ole määritelty konversiofunktioita ja useat muodostajista, jotka saavat parametrinaan yhden argumentin, on esitelty eksplisiittisiksi. Kuitenkin on määritelty monia ylikuormitettuja operaattoreita C++-vakiokirjaston luokille. Mitä luulet, miksi tällainen suunnittelupäätös on tehty?

Harjoitus 15.13

Miksi SmallInt-luokan ylikuormitettua syöttöoperaattoria, joka määritettiin tämän kohdan alussa, ei ole toteutettu seuraavaan tapaan?

```
istream& operator>>( istream &is, SmallInt &si )
{
    return ( is >> si.value );
}
```

Harjoitus 15.14

Osoita mahdolliset käyttäjän määrittelemät konversiosarjat jokaiselle seuraavalle alustukselle. Mitä on kunkin alustuksen tuloksena?

```
class LongDouble {
    operator double();
    operator float();
};

extern LongDouble ldObj;

(a) int ex1 = ldObj;
(b) float ex2 = ldObj;
```

Harjoitus 15.15

Nimeä kolme funktioehdokasjoukkoa, jotka otetaan huomioon funktion ylikuormituksen ratkaisussa, kun funktion argumenttina on luokkatyyppi.

Harjoitus 15.16

Mikä calc()-funktio, jos mikään, valitaan parhaiten elinkelpoiseksi funktioksi seuraavaa kutsua varten? Osoita tarvittavat konversiosarjat kutsua varten ja perustele, miten parhaiten elinkelpoinen funktio on valittu.

```
class LongDouble {
public
    LongDouble( double );
    // ...
};

extern void calc( int );
extern void calc( LongDouble );
double dval;

int main() {
    calc( dval ); // mikä funktio?
}
```

15.11 Ylikuormituksen ratkaisu ja jäsenfunktiot

Myös jäsenfunktioita voidaan ylikuormittaa. Funktion ylikuormituksen ratkaisua käytetään myös parhaiten elinkelpoisen funktion valintaan jäsenfunktion kutsua varten. Jäsenfunktioiden ylikuormituksen ratkaisu on paljon samanlainen kuin sellaisten funktioiden ylikuormituksen ratkaisu, jotka eivät ole jäsenfunktioita. Prosessi muodostuu samoista kolmesta vaiheesta:

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

On olemassa pieniä eroavaisuuksia siinä, kuinka funktioehdokkaat ja elinkelpoiset funktiot valitaan jäsenfunktioiden kutsuja varten. Tutkimme tässä kohdassa näitä eroavaisuuksia.

15.11.1 Ylikuormitettujen jäsenfunktioiden esittelyt

Luokan jäsenfunktioita voidaan ylikuormittaa. Esimerkiksi:

```
class myClass {
public:
    void f( double );
```

```
    char f( char, char ); // ylikuormittaa funktiota myClass::f( double )
    // ...
};
```

Kuten nimiavaruuden viittausalueelle esitellyille funktiolle, niin myös luokkaan esitellyille jäsenfunktioille voidaan antaa sama nimi edellyttäen, että parametriluettelo on yksilöllinen, joko parametrien tyypiltä tai määrältä. Jos kahden samannimisen jäsenfunktion esittelyt eroavat toisistaan vain paluutyypiltään, pidetään toista esittelyä virheellisenä esittelynä, ja aiheutuu käännösvirhe. Esimerkiksi:

```
class myClass {
public:
    void mf();
    double mf(); // virhe: ei ole kelvollinen ylikuormitettu funktio
    // ...
};
```

Kuitenkin toisin kuin nimiavaruuden funktioita, jäsenfunktiota saa esitellä vain kerran luokan jäsenluettelossa. Jos kahden samannimisen jäsenfunktion paluutyyppi ja parametriluettelo vastaavat täysin toisiaan, saa toinen esittely aikaan käännösvirheen kelpaamattomana funktion uudelleen esittelynä. Esimerkiksi:

```
class myClass {
public:
    void mf();
    void mf(); // virhe: kelvoton uudelleen esittely
    // ...
};
```

Kaikki ylikuormitetut funktiot esitellään samalla viittausalueella. Tästä syystä jäsenfunktiot eivät ylikuormita koskaan funktioita, jotka on esitelty nimiavaruuden viittausalueella. Koska myös jokainen luokka pitää yllä omaa viittausaluettaan, funktiot, jotka ovat kahden eri luokan jäseniä, eivät ylikuormita toinen toistaan.

Ylikuormitettujen funktioiden joukko voi sisältää sekä staattisia että ei-staattisia jäsenfunktioita. Esimerkiksi:

```
class myClass {
public:
    void mcf( double );
    static void mcf( int* ); // ylikuormittaa funktiota myClass::mcf( double )
    // ...
};
```

Se, kutsutaanko staattista vai ei-staattista jäsenfunktiota, riippuu funktion ylikuormituksen ratkaisun tuloksesta. Katsomme tarkemmin funktion ylikuormituksen ratkaisua seuraavassa kohdassa, kun sekä staattinen että ei-staattinen jäsenfunktio ovat elinkelpoisia funktioita.

15.11.2 Funktioehdokkaat

Jos jäsenfunktion kutsu on jompikumpi seuraavista muodoista

```
mc.mf( arg );
pmc->mf( arg );
```

ja jos mc on myClass-tyyppinen lauseke ja pmc on lauseke tyyppiä osoitin myClass-tyyppiin, jäsenfunktioehdokkaat molemmille näille kutsuille muodostuvat funktioista, jotka löytyvät myClass-luokan viittausalueelta, kun etsitään esittelyä mf()-funktiolle.

Samalla tavalla, jos funktion kutsu on muotoa

```
myClass::mf( arg );
```

funktioehdokkaitten joukko muodostuu niistä, jotka löytyvät etsittäessä esittelyjä mf()-funktiolle myClass-luokan viittausalueelta. Esimerkiksi:

```
class myClass {
public:
    void mf( double );
    char mf( char, char = '\n' );
    static void mf( int* );
    // ...
};

int main() {
    myClass mc;
    int iobj;
    mc.mf( iobj );
}
```

Funktioehdokkaat funktiokutsulle, joka tehdään main()-funktiossa, ovat kolme mf()-jäsenfunktiota, jotka on esitelty myClass-luokassa:

```
void mf( double );
char mf( char, char = '\n' );
static void mf( int* );
```

Ellei mf()-nimistä jäsenfunktiota löydy myClass-luokasta, on funktioehdokkaitten joukko tyhjä. (Todellisuudessa otetaan huomioon funktiot kantaluokissa. Se, kuinka kantaluokan jäsenfunktiot tulevat funktioehdokkaitten joukkoon, käsitellään kohdassa 19.3). Ellei funktioehdokkaita löydy funktion kutsua varten, kutsu saa aikaan käännösvirheen.

15.11.3 Elinkelpoiset funktiot

Elinkelpoinen funktio on jäsenfunktioehdokkaiden joukosta sellainen, jota voidaan kutsua kutsussa määritetyllä argumenttiluettelolla. Se on funktio, jolle löytyy implisiittisiä tyyppikonversioita argumenttien ja funktion parametrien tyyppien välille. Esimerkiksi:

```
class myClass {
public:
```

```
    void mf( double );
    char mf( char, char = '\n' );
    static void mf( int* );
    // ...
};

int main() {
    myClass mc;
    int iobj;
    mc.mf( iobj ); // mikä mf()-jäsenfunktio? se on moniselitteinen
}
```

On olemassa kaksi elinkelpoista funktiota mf()-jäsenfunktion kutsua varten main()-funktiossa:

```
void mf( double );
char mf( char, char = '\n' );
```

1. mf(double) on elinkelpoinen jäsenfunktio, koska sillä on vain yksi parametri ja koska löytyy vakiokonversio, jolla voidaan konvertoida int-tyyppinen iobj-argumentti double-tyyppiseksi parametriksi.
2. mf(char, char) on elinkelpoinen jäsenfunktio, koska siinä on oletusargumentti toiselle parametrille ja koska löytyy vakiokonversio, jolla voidaan konvertoida int-tyyppinen iobj-argumentti ensimmäisen parametrin char-tyypiksi.

Kun parhaiten elinkelpoista jäsenfunktiota valitaan, laitetaan jokaisen argumentin tyyppikonversiot paremmuusjärjestykseen. Parhaiten elinkelpoinen jäsenfunktio on se, jonka argumentteihin käytetyt konversiot *eivät ole huonompia* kuin konversiot, jotka ovat tarpeen mille tahansa muulle elinkelpoiselle funktiolle, ja joidenkin argumenttien konversiot ovat *parempia* kuin konversiot, jotka ovat tarpeen samoille argumenteille, kun kutsutaan mitä tahansa muuta elinkelpoista funktiota.

Edellisessä esimerkissä konversio, jota käytetään argumentille, jotta se vastaisi molempien elinkelpoisten jäsenfunktioiden parametreja, on vakiokonversio. Kutsu on siten moniselitteinen, koska molemmat funktiot ovat yhtä hyviä funktion kutsussa määritellylle argumentille.

Funktion muodosta riippumatta sekä staattisia että ei-staattisia jäsenfunktioita voidaan ottaa mukaan elinkelpoisten funktioiden joukkoon. Esimerkiksi:

```
class myClass {
public:
    static void mf( int );
    char mf( char );
};

int main() {
    char cobj;
    myClass::mf( cobj ); // mikä jäsenfunktio?
}
```

Vaikka mf()-jäsenfunktiota kutsutaan luokan nimeä ja viittausalueen erotteluoperaattoria (myClass::mf()) käyttäen ja vaikka mf()-jäsenfunktiota ei kutsuta olion tai olio-osoittimen kautta käyttäen luokan jäsenen käsittelyoperaattoria, pistettä tai nuolta, otetaan ei-staattinen mf(char)-jäsenfunktio mukaan elinkelpoisiin funktioihin kutsua varten, kuten myös staattinen mf(int)-jäsenfunktio.

Funktion ylikuormituksen ratkaisu etenee sitten parhaiten elinkelpoisen funktion valintaan laittamalla funktion argumenttien tyyppikonversiot paremmuusjärjestykseen. Argumentti cobj, joka on char-tyyppi, vastaa täysin mf(char)-jäsenfunktion parametria. Argumentti voidaan konvertoida mf(int)-jäsenfunktion parametrin tyypiksi ylennyksen kautta. Kun katsotaan tässä esimerkissä argumenteille käytettyjä konversioita, valitaan mf(char)-jäsenfunktio parhaiten elinkelpoiseksi funktioksi.

Kuitenkin valittu jäsenfunktio on ei-staattinen, eikä sitä voi kutsua suoraan. Sitä pitää kutsua olion tai olio-osoittimen kautta, joka on myClass-tyyppinen ja joka käyttää jompaakumpaa jäsenen käsittelyoperaattoria, pistettä tai nuolta. Mitä sitten tapahtuu? Jos parhaiten elinkelpoiseksi valittu funktio on ei-staattinen jäsenfunktio eikä kutsu voi todellisuudessa tapahtua, koska oliota ei ole määritetty kutsua varten (kuten tilanne on tässä), kääntäjä ilmoittaa kutsun olevan virheellinen.

Toinen näkökohta jäsenfunktioista, jotka pitää ottaa huomioon elinkelpoisia funktioita valittaessa, on ei-staattisten jäsenfunktioiden const- tai volatile-määreet. (const- ja volatile-jäsenfunktiot esiteltiin kohdassa 13.3). Kuinka nämä piirteet vaikuttavat funktion ylikuormituksen ratkaisuun? Jos myClass-luokassa on seuraavat jäsenfunktiot

```
class myClass {
public:
    static void mf( int* );
    void mf( double );
    void mf( int ) const;
    // ...
};
```

staattinen jäsenfunktio mf(int*), const-jäsenfunktio mf(int) ja ei-const-jäsenfunktio mf(double) otetaan kaikki mukaan funktioehdokkaitten joukkoon seuraavaa kutsua varten. Mitkä jäsenfunktiot otetaan mukaan elinkelpoisiin funktioihin?

```
int main() {
    const myClass mc;
    double dobj;
    mc.mf( dobj ); // mikä mf()-jäsenfunktio?
}
```

Kun tutkitaan konversioita, joita voidaan käyttää funktion argumenttiin, ovat mf(double) ja mf(int) elinkelpoisia funktioita. double-tyyppinen dobj-argumentti vastaa täysin mf(double)-jäsenfunktion parametria. dobj-argumentti voidaan konvertoida mf(int)-jäsenfunktion parametrin tyyppiseksi vakiokonversion kautta.

Kun jäsenfunktion kutsussa käytetään jäsenen käsittelyoperaattoria, pistettä tai nuolta, otetaan huomioon olion tai osoittimen tyyppi, jonka kautta jäsenfunktiota kutsutaan, kun valitaan funktioita elinkelpoisiin funktioihin.

mc on const-olio. Vain const-tyyppisiä ei-staattisia jäsenfunktioita voidaan kutsua const-olioille. Koska ei-const-tyyppistä ja ei-staattista mf(double)-jäsenfunktiota ei voi kutsua, jätetään se pois elinkelpoisten funktioiden joukosta. Ainoa elinkelpoinen funktio tälle kutsulle on const-jäsenfunktio mf(int), joka valitaan kutsua varten.

Mitä, jos const-oliota käytetään staattisen jäsenfunktion kutsussa? Koska staattisia jäsenfunktioita ei voida esitellä const- tai volatile-määreellä, voidaako sitä kutsua const-olion kautta? Esimerkiksi:

```
class myClass {
public:
    static void mf( int );
    char mf( char );
};
int main() {
    const myClass mc;
    int iobj;
    mc.mf( iobj ); // voidaako staattista funktiota kutsua?
}
```

Staattiset jäsenfunktiot ovat geneerisiä kaikille tietyntyyppisille olioille. Staattiset jäsenfunktiot voivat käsitellä vain luokan staattisia jäseniä suoraan. Esimerkiksi staattinen mf(int)-jäsenfunktio ei voi käsitellä const-tyyppisen mc-olion ei-staattisia jäseniä. Tästä syystä on aina sallittua kutsua staattista jäsenfunktiota const-oliolla käyttämällä piste- tai nuolioperaattoria.

Täten staattisia jäsenfunktioita ei koskaan jätetä pois elinkelpoisten funktioiden joukosta olioiden tai osoittimien const- tai volatile-määreiden vuoksi, joiden kautta niitä kutsutaan. Staattisten jäsenfunktioiden katsotaan vastaavan kaikkia luokkansa tyyppisiä olioita tai osoittimia.

Koska edellisessä esimerkissä mc on const-olio, jätetään jäsenfunktio mf(char) pois elinkelpoisten funktioiden joukosta. Kuitenkin jäsenfunktio mf(int) otetaan mukaan elinkelpoisten funktioiden joukkoon, koska se on staattinen funktio. Koska se on ainoa elinkelpoinen funktio kutsua varten, valitaan se parhaiten elinkelpoiseksi funktioksi.

15.12 Ylikuormituksen ratkaisu ja operaattorit ◆A

Kuten olemme nähneet edellisissä kohdissa, luokkatyypeille voidaan esitellä ylikuormitettuja operaattoreita ja konversiofunktioita. Kuinka kääntäjä päättää kohdatessaan operaattorin kuten lisäysoperaattorin seuraavassa alustuksessa

```
SomeClass sc;
int iobj = sc + 3;
```

tulisiko sen käyttää ylikuormitettua lisäysoperaattoria SomeClass-luokalle vai konversiofunktiota sc-operandin konvertoimiseksi sisäiseksi tyypiksi ja käyttää sitten sisäistä lisäysoperaattoria?

Vastaus riippuu SomeClass-luokkaan määritellyistä ylikuormitetuista operaattoreista ja konversiofunktioista. Funktion ylikuormituksen ratkaisun prosessia käytetään sen operaattorin valitsemiseen, joka suorittaa yhteenlaskun. Tässä kohtaa katsomme, kuinka ylikuormituksen ratkaisu etenee operaattorien valinnassa, kun niitä käytetään luokkatyyppien operandeina.

Ylikuormituksen ratkaisu ylikuormitetuille operaattoreille noudattaa kolmevaiheista prosessia, joka esiteltiin kohdassa 9.2:

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

Tutkimme näitä kolmea vaihetta tarkemmin tässä kohdassa.

Funktion ylikuormituksen ratkaisua ei koskaan käytetä, jos operaattorilla on vain sisäisten tyyppien operandeja. Sellaisille operandeille on taattu sisäisen operaattorin käyttö. (Operaattorien käyttö sisäisten tyyppien yhteydessä on kuvattu luvussa 4). Esimerkiksi:

```
class SmallInt {
public:
    SmallInt( int );
};

SmallInt operator+ ( const SmallInt &, const SmallInt & );

void func() {
    int i1, i2;
    int i3 = i1 + i2;
}
```

Koska i1 ja i2 ovat int-tyyppisiä operandeja eivätkä luokkatyyppejä, käytetään sisäistä +-operaattoria yhteenlaskuun i1 + i2. Ylikuormitettu operaattori operator+(const SmallInt&, const SmallInt&) jätetään huomiotta, vaikka operandit voitaisiin konvertoida SmallInt-tyyppisiksi käyttäjän määrittelemän konversion kautta käynnistämällä SmallInt(int)-muodostaja. Tässä kuvattua ylikuormituksen ratkaisua ei käytetä sellaisissa tilanteissa.

Tässä kuvattu ylikuormituksen ratkaisun prosessi operaattoreille pätee vain, kun käytetään operaattorisyntaksia. Esimerkiksi:

```
void func() {
    SmallInt si(98);
    int iobj = 65;
    int res = si + iobj; // operaattorisyntaksia käytetty
}
```

Jos käytetään sen sijaan funktion kutsun syntaksia kuten tässä

```
int res = operator+( si , iobj ); // funktion kutsun syntaksia käytetty
```

silloin käytetään ylikuormituksen ratkaisua nimiavaruuden funktioille, kuten kohdassa 15.10 kuvattiin. Jos käytetään sen sijaan jäsenfunktion kutsusyntaksia, kuten tässä

```
// tässä on käytetty jäsenen kutsusyntaksia
int res = si.operator+( iobj );
```

silloin käytetään ylikuormituksen ratkaisua luokan jäsenfunktioille, kuten kohdassa 15.11 kuvattiin.

15.12.1 Operaattorifunktioehdokkaat

Operaattorifunktioehdokas on sellainen, jolla on sama nimi kuin kutsutulla funktiolla. Seuraavalle lisäysoperaattorille

```
SmallInt si(98);
int iobj = 65;
int res = si + iobj;
```

on operaattorifunktioehdokas nimeltään operator+. Mitkä operator+()-esittelyt otetaan huomioon?

Potentiaalisesti on mahdollista, että on rakennettu *viisi* operaattorifunktioehdokasjoukkoa operaattorin käyttöön käyttämällä operaattorisyntaksia luokkatyyppisen operandin kanssa. Kolme ensimmäistä joukkoa ovat samoja kuin ne, jotka on muodostettu tavallisille funktiokutsuille luokkatyyppisin argumentein:

1. Operaattorit, jotka ovat näkyvissä kutsupaikassa. Operaattorin operator+() esittelyt, jotka ovat näkyvissä, kun operaattoria käytetään, ovat operaattorifunktioehdokkaita. Esimerkiksi operator+()-operaattori, joka on esitelty globaalilla viittausalueella, on funktioehdokas operator+()-operaattorin käyttötilanteessa main()-funktiossa:

   ```
   SmallInt operator+ ( const SmallInt &, const SmallInt & );

   int main() {
       SmallInt si(98);
       int iobj = 65;
       int res = si + iobj; // ::operator+() on funktioehdokas
   }
   ```

2. Operaattorijoukko, joka on määritelty nimiavaruuteen, jossa operandin tyyppi on määritelty. Jos operandi on luokkatyyppinen ja tyyppi on esitelty käyttäjän määrittelemässä nimiavaruudessa, tuossa nimiavaruudessa esitellyt operaattorifunktiot, jotka ovat samannimisiä kuin operaattori, ovat operaattorifunktioehdokkaita. Esimerkiksi:

   ```
   namespace NS {
       class SmallInt { /* ... */ };
       SmallInt operator+ ( const SmallInt&, double );
   }
   ```

```
int main() {
    // si:n tyyppi on SmallInt-luokka:
    // luokka on esitelty NS-nimiavaruudessa
    NS::SmallInt si(15);

    // NS::operator+() on funktioehdokas
    int res = si + 566;
    return 0;
}
```

si-operandin tyyppi on SmallInt-luokkatyyppi, joka on esitelty NS-nimiavaruudessa. Ylikuormitusoperaattori, operator+(const SmallInt&, double), joka on esitelty NS-nimiavaruudessa, lisätään operaattorifunktioehdokkaitten joukkoon.

3. Operaattorit, jotka on esitelty operandin luokkatyypin ystävinä. Jos luokkatyyppisillä operandeilla luokan määrittelyssä esitellään samannimisiä ystäväoperaattorifunktioita kuin käytetty operandi on, lisätään ystäväoperaattorifunktiot operaattorifunktioehdokkaitten joukkoon. Esimerkiksi:

```
namespace NS {
    class SmallInt {
        friend SmallInt operator+(const SmallInt&, int )
                        { /* ... */ }
    };
}

int main() {
    NS::SmallInt si(15);

    // ystäväoperaattori operator+() on ehdokas
    int res = si + 566;
    return 0;
}
```

si-operandin tyyppi on SmallInt. Sen ystäväoperaattorifunktio, operator+(const SmallInt&, int), on NS-nimiavaruuden jäsen, vaikka sitä ei koskaan ole esitelty NS-nimiavaruudessa suoraan. Normaalietsintä NS-nimiavaruudesta ei löydä ystäväoperaattorifunktiota. Kuitenkin, kun operator+()-operaattoria käytetään argumentilla, joka on SmallInt-tyyppinen, otetaan SmallInt-luokan viittausalueella esitellyt ystäväfunktiot huomioon ja lisätään funktioehdokkaiden joukkoon.

Edelliset kolme operaattorifunktioehdokkaitten joukkoa saatetaan loppuun samalla tavalla kuin funktioehdokkaat funktiokutsuissa, jossa argumentit ovat luokkatyyppisiä. Kuitenkin operaattorit, joita käytetään operaattorisyntaksilla, saatetaan loppuun kahdella muulla operaattorifunktioehdokasjoukolla, joka tekee yhteensä viisi operaattorifunktioehdokasjoukkoa:

4. Jäsenoperaattorit, jotka on esitelty vasemmanpuoleisen operandin luokassa. Jos operator+()-operaattoria on käytetty vasemmanpuoleisessa luokkatyyppisessä operandissa,

jäsenoperaattoriehdokkaat rakennetaan etsimällä esittelyitä operator+()-jäsenelle vasemmanpuoleisen operandin luokasta. Esimerkiksi:

```
class myFloat {
    myFloat( double );
};
class Smallint {
public:
    SmallInt( int );
    SmallInt operator+ ( const myFloat & );
};

int main() {
    SmallInt si(15);

    int res = si + 5.66; // jäsenoperaattori operator+() on ehdokas
}
```

Jäsenoperaattori SmallInt::operator+(const myFloat &), joka on määritelty SmallInt-luokassa, otetaan mukaan funktioehdokkaitten joukkoon operator+()-operaattorin kutsussa main()-funktiossa.

5. Sisäisten operaattorien joukko. Olettaen, että tyypit, joita voidaan käyttää sisäisen operator+()-operaattorin kanssa, ovat operaattorifunktioehdokkaat sisäiselle binääriselle operator+()-operaattorille seuraavat:

```
int operator+( int, int )
double operator+( double, double )
T* operator+( T*, I )
T* operator+( I, T* )
```

Ensimmäinen esittely edustaa sisäistä operaattoria, jota voidaan käyttää minkä tahansa kahden kokonaistyypin yhteenlaskemiseen. Toinen esittely edustaa sisäistä operaattoria, jota voidaan käyttää minkä tahansa kahden liukulukutyypin yhteenlaskemiseen. Kolmas ja neljäs esittely edustavat sellaista sisäistä operaattoria osoitintyypeille, jota käytetään kokonaistyyppien arvojen lisäämiseen osoitintyyppiarvoihin. Nämä esittelyt ovat vain symbolisia. Niitä käytetään kuvaamaan sisäisiä operaattoriehdokkaita, jotka kääntäjä ottaa huomioon kaikissa yhteenlaskuoperaatioissa.

Kun muodostetaan neljää ensimmäistä operaattorifunktioehdokkaitten joukkoa, on mahdollista, että ehdokasjoukosta tulee tyhjä. Jos esimerkiksi jäsenfunktiota nimeltään operator+() ei löydy SmallInt-luokasta, operaattorifunktioehdokkaitten joukko (tarkoittaa neljättä joukkoa) on tyhjä.

Operaattorifunktioehdokkaitten joukko on aikaisemmin lueteltujen viiden funktioehdokasjoukon yhdiste. Esimerkiksi:

```
namespace NS {
    class myFloat {
```

```
        myFloat( double );
    };
    class SmallInt {
        friend SmallInt operator+(const SmallInt &, int ) { /* ... */ }
    public:
        SmallInt( int );
        operator int();
        SmallInt operator+( const myFloat & );
        // ...
    };
    SmallInt operator+( const SmallInt &, double );
}

int main() {
    // si:n tyyppi on SmallInt-luokka:
    // luokka on esitelty NS-nimiavaruudessa
    NS::SmallInt si(15);

    int res = si + 5.66; // mikä operaattori: operator+ ?
    return 0;
}
```

Viisi funktioehdokasjoukkoa antavat tuloksena seitsemän operaattorifunktioehdokasta operator+()-operaattorin kutsuun main()-funktiossa:

1. Ensimmäinen funktioehdokasjoukko on tyhjä. Ylikuormitetulle operator+()-operaattorille ei ole näkyvissä esittelyitä globaalilla viittausalueella, jossa operator+()-operaattoria on käytetty main()-funktiossa.
2. Toinen funktioehdokasjoukko sisältää operaattorit, jotka on esitelty NS-nimiavaruudessa, jossa SmallInt-luokkatyyppi on määritelty. Seuraava operaattori on määritelty NS-nimiavaruudessa:

   ```
   NS::SmallInt NS::operator+( const SmallInt &, double );
   ```

3. Kolmas funktioehdokasjoukko sisältää operaattorit, jotka on esitelty SmallInt-luokan ystävinä. Seuraava operaattori on SmallInt-luokan ystävä:

   ```
   NS::SmallInt NS::operator+( const SmallInt &, int );
   ```

4. Neljäs funktioehdokasjoukko sisältää operaattorit, jotka on esitelty SmallInt-luokan jäseninä. Seuraava operaattori on SmallInt-luokan jäsen:

   ```
   NS::SmallInt NS::SmallInt::operator+( const myFloat & );
   ```

5. Viides funktioehdokasjoukko sisältää sisäiset binääriset operaattorit:

   ```
   int operator+( int, int )
   double operator+( double, double )
   T* operator+( T*, I )
   T* operator+( I, T* )
   ```

Huh! Kyllä, operaattorisyntaksia käyttävän operaattorin funktioehdokkaitten joukon läpikäyminen voi osoittautua melkoisen työlääksi. Kun funktioehdokkaitten joukko on muodostettu, elinkelpoiset funktiot ja niistä parhaiten elinkelpoinen funktio löytyvät kuten ennenkin analysoimalla konversioita, joita voidaan tehdä ehdokasoperaattorin operandeille.

15.12.2 Elinkelpoiset funktiot

Elinkelpoisten operaattorifunktioiden joukko valitaan operaattorifunktioehdokkaitten joukosta valitsemalla vain ne operaattorifunktiot, joita voidaan kutsua operandeilla, jotka on määritetty operaattoria käytettäessä. Esimerkiksi, mitkä esimerkkimme seitsemästä funktioehdokkaasta ovat elinkelpoisia funktioita? Operaattorin käyttötilanne on kuten seuraavassa:

```
NS::SmallInt si(15);
si + 5.66;
```

Vasemmanpuoleinen operandi on SmallInt-tyyppi ja oikeanpuoleinen on double-tyyppi.

Ensimmäinen funktioehdokas on elinkelpoinen operator+()-operaattorin käytön takia:

```
NS::SmallInt NS::operator+( const SmallInt &, double );
```

Vasemmanpuoleinen SmallInt-tyyppinen si-operandi vastaa täysin ylikuormitetun operaattorin viittausparametria alustajana. Oikeanpuoleinen operandi on double-tyyppinen arvo, joka myös vastaa täysin ylikuormitetun operaattorin toista parametria.

Toinen funktioehdokas on myös elinkelpoinen operator+()-operaattorin käytön takia:

```
NS::SmallInt NS::operator+( const SmallInt &, int );
```

Vasemmanpuoleinen SmallInt-tyyppinen si-operandi vastaa täysin ylikuormitetun operaattorin viittausparametria sen alustajana. Oikeanpuoleinen operandi on int-tyyppinen arvo, joka voidaan konvertoida ylikuormitetun operaattorin toiseksi parametriksi vakiokonversion avulla.

Kolmas funktioehdokas on myös elinkelpoinen operator+()-operaattorin käytön takia:

```
NS::SmallInt NS::SmallInt::operator+( const myFloat & );
```

Vasemmanpuoleinen si-operandi on SmallInt-tyyppiä, joka on luokkatyyppi, jossa ylikuormitettu operaattori on määritelty jäsenfunktiona. Oikeanpuoleinen operandi on int-tyyppinen arvo, joka voidaan konvertoida myFloat-luokkatyypiksi käyttäjän määrittelemän konversiosarjan kautta käyttämällä myFloat(double)-muodostajaa.

Neljäs ja viides elinkelpoinen funktio ovat sisäisiä operaattoreita:

```
int operator+( int, int )
double operator+( double, double )
```

SmallInt-luokka sisältää konversiofunktion, jolla voidaan konvertoida SmallInt-tyyppinen arvo int-tyypiksi. Konversiofunktiota käytetään ensimmäiselle sisäiselle operaattorille konvertoitaessa vasen SmallInt-tyyppinen operandi int-tyypiksi. Toinen double-tyyppinen operandi konvertoidaan int-tyypiksi vakiokonversiolla. Konversiofunktiota käytetään toiselle sisäiselle

operaattorille konvertoitaessa vasemmanpuoleinen SmallInt-tyyppinen operandi int-tyypiksi ja sen tulos konvertoidaan sitten double-tyypiksi vakiokonversion kautta. Toinen double-tyyppinen operandi vastaa täysin toista parametria.

Parhaiten elinkelpoinen funktio viidestä elinkelpoisesta funktiosta on niistä ensimmäinen, ylikuormitettu operator+()-operaattori, joka on esitelty NS-nimiavaruudessa:

```
NS::SmallInt NS::operator+ ( const SmallInt &, double );
```

Molemmat operandit vastaavat täysin tämän ylikuormitetun operaattorin parametreja.

15.12.3 Moniselitteisyys

Sekä konversiofunktioiden, jotka suorittavat implisiittiset konversiot sisäisille tyypeille, että ylikuormitettujen operaattorien tekeminen samalle luokkatyypille voi johtaa moniselitteisyyksiin ylikuormitettujen operaattoreiden ja sisäisten operaattoreiden välillä. Jos esimerkiksi String-luokkaan on määritelty seuraava vertailufunktio

```
class String {
    // ...
public:
    String( const char * = 0 );
    bool operator== ( const String & ) const;
    // operaattoria operator== ( const char * ) ei ole tehty
};
```

ja seuraava operator==()-operaattorin käyttötilanne

```
String flower( "tulip" );

void foo( const char *pf ) {
    // kutsuu ylikuormitettua operaattoria String::operator==()
    if ( flower == pf )
        cout << pf << " is a flower!\en";
    // ...
}
```

vertailu käynnistää

```
flower == pf
```

String-luokan yhtäsuuruusjäsenoperaattorin

```
String::operator==( const String & ) const;
```

Käyttäjän määrittelemää konversiota, joka kutsuu muodostajaa

```
String( const char * )
```

käytetään konvertoitaessa oikeanpuoleinen pf-operandi const char* -tyypistä String-tyypiksi, joka on operator==()-jäsenen parametrin tyyppi.

Jos konversiofunktion const char*() -operaattori lisätään String-luokan määrittelyyn

```
class String {
    // ...
```

```
public:
    String( const char * = 0 );
    bool operator== ( const String & ) const;
    operator const char*(); // uusi konversiofunktio
};
```

on aikaisempi operator==()-operaattorin käyttötilanne nyt moniselitteinen

```
// yhtäsuuruustesti ei enää käänny!
if ( flower == pf )
```

Konversiofunktion const char*() -operaattorin takia sisäinen vertailuoperaattori

```
bool operator==( const char *, const char * )
```

on nyt myös elinkelpoinen funktio. Vasemmanpuoleinen String-tyyppinen flower-operandi voidaan konvertoida const char* -tyypiksi käyttäen uutta käyttäjän määrittelemää konversiota.

Nyt on kaksi elinkelpoista operaattorifunktiota operator==()-operaattorin käyttöön foo()-funktiossa. Ensimmäinen elinkelpoinen funktio

```
String::operator==( const String & ) const;
```

vaatii käyttäjän määrittelemän konversion, jolla konvertoidaan oikeanpuoleinen pf-operandi const char* -tyypistä String-tyypiksi. Toinen elinkelpoinen funktio

```
bool operator==( const char * , const char * )
```

vaatii käyttäjän määrittelemän konversion, jolla konvertoidaan vasemmanpuoleinen String-tyyppinen flower-operandi const char* -tyypiksi.

Ensimmäinen elinkelpoinen funktio on parempi vasemmanpuoleiselle operandille, kun taas toinen elinkelpoinen funktio on parempi oikeanpuoleiselle operandille. Kutsu saa siten aikaan käännösvirheen moniselitteisyytensä takia, koska parhaiten elinkelpoista funktiota ei löydy.

Tästä syystä pitää olla varovainen, kun suunnitellaan luokalle rajapintaa ja kun esitellään ylikuormitettuja operaattoreita, muodostajia ja konversiofunktioita tietylle luokkatyypille. Kääntäjä käyttää käyttäjän määrittelemiä konversioita implisiittisesti. Tämä voi saada aikaan, että sisäisistä operaattoreista tulee elinkelpoisia funktioita luokkatyyppisille operandeille operaattoreiden käytön yhteydessä. Konversiofunktioita ja ei-eksplisiittisiä muodostajia tulisi siksi käyttää järkevästi.

Harjoitus 15.17

Nimeä viisi funktioehdokasjoukkoa, jotka otetaan huomioon funktion ylikuormituksen ratkaisun aikana, jos operaattoria on käytetty luokkatyyppisen operandin kanssa.

Harjoitus 15.18

Mikä operator+()-operaattori, jos mikään, valitaan parhaiten elinkelpoiseksi funktioksi yhteenlaskuoperaatiolle main()-funktiossa? Luettele funktioehdokkaat, elinkelpoiset funktiot ja jokaisen elinkelpoisen funktion argumenttien tyyppikonversiot.

```
namespace NS {
    class complex {
        complex( double );
        // ...
    };
    class LongDouble {
        friend LongDouble operator+( LongDouble &, int ) { /*...*/ }
    public:
        LongDouble( int );
        operator double();
        LongDouble operator+( const complex & );
        // ...
    };
    LongDouble operator+( const LongDouble &, double );
}

int main() {
    // si:n tyyppi on SmallInt-luokka:
    // luokka on esitelty NS-nimiavaruudessa
    NS:: LongDouble ld(16.08);

    double res = ld + 15.05; // mikä operaattori: operator+ ?
    return 0;
}
```