Osa
IV

14. oppitunti

Operaattorin ylikuormitus

Edellisessä luvussa opit ylikuormittamaan metodeita ja luomaan kopiomuodostimen, joka tekee syvän kopion. Tässä luvussa käsitellään seuraavia aiheita:

- Kuinka jäsenfunktioita ylikuormitetaan
- Kuinka sijoitusoperaattori ylikuormitetaan hallitsemaan muistia
- Kuinka luodaan funktioita, jotka tukevat luokkia, joissa on dynaamisesti varattuja muuttujia

Operaattorin ylikuormittaminen

C++ -kielessä on lukuisia sisäisiä tietotyyppejä, kuten int, real, char, jne. Jokaisella noista tyypeistä on joukko sisäisiä operaattoreita, kuten

yhteenlasku (+), kertolasku (*), jne. C++ mahdollistaa noiden operaattoreiden lisäämisen myös omiin luokkiin.

Tutkiaksemme operaattorin ylikuormittamista luodaan listauksessa 14.1 uusi luokka, Counter. Counter-oliota käytetään laskemiseen (yllätys!) silmukoissa sekä muissa sovelluksissa, joissa lukua kasvatetaan, vähennetään tai muokataan muulla tavalla.

Listaus 14.1. Counter-luokka.

```
1:  // Listaus 14.1
2:  // Counter-luokka
3:
4:  #include <iostream.h>
5:
6:  class Counter
7:  {
8:  public:
9:
10:   Counter();
11:   ~Counter(){}
12:   int GetItsVal()const { return itsVal; }
13:   void SetItsVal(int x) {itsVal = x; }
14:
15: private:
16:   int itsVal;
17:
18: };
19:
20: Counter::Counter():
21: itsVal(0)
22: {};
23:
24: int main()
25: {
26:   Counter i;
27:   cout << "The value of i is " << i.GetItsVal() << endl;
28:   return 0;
29: }
```

Tulostus

```
The value of i is 0.
```

Analyysi

Kyseessä on melko hyödytön luokka. Se määritellään riveillä 7-18. Sen ainoa jäsenmuuttuja on int. Rivillä 10 on esitelty oletusmuodostin ja sen toteutus on rivillä 20. Se alustaa jäsenmuuttujan, itsVal, arvolla 0.

Päinvastoin kuin aitoa int-muuttujaa, ei laskurioliota voida kasvattaa, vähentää, käyttää yhteenlaskuun tai sijoitukseen eikä muutenkaan muokata. Tämä tekee myös sen arvon tulostamisen vaikeaksi.

Kasvatusfunktion muodostaminen

Operaattorin ylikuormittaminen tuo paljon toiminnallisuutta, johon käyttäjän määrittelemä luokka ei itsessään kykene. Listaus 14.2 havainnollistaa kasvatusoperaattorin ylikuormittamista.

Listaus 14.2. Kasvatusoperaattorin ylikuormittaminen.

```
1:  // Inkrementti-operaattorin ylimäärittely
2:
3:  typedef unsigned short  USHORT;
4:  #include <iostream.h>
5:
6:
7:  class Counter
8:  {
9:  public:
10:   Counter();
11:   ~Counter(){}
12:   USHORT GetItsVal()const { return itsVal; }
13:   void SetItsVal(USHORT x) {itsVal = x; }
14:   void Increment() { ++itsVal; }
15:   const Counter& operator++ ();
16:
17: private:
18:   USHORT itsVal;
19:
20: };
21:
22: Counter::Counter():
23: itsVal(0)
24: {};
25:
26: const Counter& Counter::operator++()
27: {
28:   ++itsVal;
29:   return *this;
30: }
31:
32: int main()
33: {
34:   Counter i;
35:   cout << "The value of i is " << i.GetItsVal() << endl;
36:   i.Increment();
37:   cout << "The value of i is " << i.GetItsVal() << endl;
38:   ++i;
39:   cout << "The value of i is " << i.GetItsVal() << endl;
40:   Counter a = ++i;
41:   cout << "The value of a: " << a.GetItsVal();
42:   cout << " and i: " << i.GetItsVal() << endl;
43:   return 0;
44: }
```

Tulostus

```
The value of i is 0
The value of i is 1
The value of i is 2
The value of a: 3 and i: 3
```

Analyysi
Operaattorin ++ toteutus riveillä 26-30 on muutettu viittaamaan this-osoittimeen ja palauttamaan nykyisen olion. This-osoitin mahdollistaa Counter-olion sijoittamisen muuttujaan a. Jos Counter-olio varaisi muistia, olisi tärkeää korvata kopiomuodostin. Tässä tapauksessa kopiomuodostin toimii hyvin.

Huomaa, että palautettu arvo on Counter-viittaus, jolloin ei luoda ylimääräistä tilapäiskopiota. Kyseessä on const-viittaus, koska tätä Counter-oliota käyttävän funktion ei tule muuttaa arvoa.

Jälkiliiteoperaattorin ylikuormittaminen

Entä, jos on tarvetta ylikuormittaa jälkiliitteenä oleva kasvatusoperaattori? Tällöin kääntäjällä on ongelmia. Kuinka se osaa erottaa etuliite- ja jälkiliite-operaattorin? Yleissäännön mukaan kokonaislukumuuttuja viedään parametrina operaattorin esittelyyn. Parametrin arvo ohitetaan; se on vain signaali siitä, että kyseessä on jälkiliiteoperaattori.

Etu- ja jälkiliitteen ero

Ennen kuin luot jälkiliiteoperaattorin sinun on ymmärrettävä, kuinka se eroaa etuliiteoperaattorista. Etuliite kertoo: kasvata ja nouda sitten, jälkiliite: nouda ja kasvata sitten.

Kun etuliiteoperaattori kasvattaa arvoa ja palauttaa sitten itse kohteen, täytyy jälkiliiteoperaattorin palauttaa arvo ennen sen kasvattamista. Tällöin on siis luotava tilapäinen kopio. Tuo tilapäiskopio tallentaa alkuperäisen arvon silloin, kun alkuperäistä kohdetta kasvatetaan. Tilapäiskopio kuitenkin palautetaan, koska jälkiliiteoperaattori käsittelee alkuperäistä arvoa.

Jos kirjoitat esimerkiksi

```
a = x++;
```

ja x oli 5, on a lauseen suorittamisen jälkeen 5, mutta x taas 6. Tällöin siis muuttujaan a sijoitetaan x:n arvo ennen sen kasvattamista. Jos x on olio, sen jälkiliiteoperaattorin täytyy sijoittaa alkuperäinen arvo tilapäiseen olioon, kasvattaa sitten x:n arvoksi 6 ja palauttaa tuo tilapäiskopio sijoitettavaksi a:han.

Huomaa, että koska tilapäiskopio palautetaan, se on palautettava arvona, ei viittauksena, koska tilapäiskopion näkyvyysalue päättyy funktion päättyessä.

Listaus 14.3. havainnollistaa etuliite- ja jälkiliiteoperaattoreiden käyttöä.

Listaus 14.3. Etuliite- ja jälkiliiteoperaattorit.

```
1:  // Listaus 14.3
2:  // Palautetaan uudelleen viittaava viittaus
3:
4:  typedef unsigned short  USHORT;
5:
6:  #include <iostream.h>
7:
8:  class Counter
9:  {
10: public:
11:   Counter();
12:   ~Counter(){}
13:   USHORT GetItsVal()const { return itsVal; }
14:   void SetItsVal(USHORT x) {itsVal = x; }
15:   const Counter& operator++ ();      // edessä
16:   const Counter operator++ (int); // jäljessä
17:
18: private:
19:   USHORT itsVal;
20: };
21:
22: Counter::Counter():
23: itsVal(0)
24: {}
25:
26: const Counter& Counter::operator++()
27: {
28:   ++itsVal;
29:   return *this;
30: }
31:
32: const Counter Counter::operator++(int)
33: {
34:   Counter temp(*this);
35:   ++itsVal;
36:   return temp;
37: }
38:
39: int main()
40: {
41:   Counter i;
42:   cout << "The value of i is " << i.GetItsVal() << endl;
43:   i++;
44:   cout << "The value of i is " << i.GetItsVal() << endl;
45:   ++i;
46:   cout << "The value of i is " << i.GetItsVal() << endl;
47:   Counter a = ++i;
48:   cout << "The value of a: " << a.GetItsVal();
49:   cout << " and i: " << i.GetItsVal() << endl;
50:   a = i++;
51:   cout << "The value of a: " << a.GetItsVal();
52:   cout << " and i: " << i.GetItsVal() << endl;
53:   return 0;
54: }
```

Tulostus

```
The value of i is 0
The value of i is 1
```

```
The value of i is 2
The value of a: 3 and i: 3
The value of a: 4 and i: 4
```

Analyysi
Jälkiliiteoperaattori esitellään rivillä 15 ja toteutetaan riveillä 32-36. Huomaa, että rivin 14 kutsussa ei käytetä lippukokonaislukua (x), vaan se on normaalissa muodossa. Jälkiliiteoperaattori käyttää lippua (x) kertomaan, että kyseessä on jälkiliite eikä etuliite. Lippuarvoa ei kuitenkaan koskaan käytetä.

Operaattori +

Kasvatusoperaattori on unaarinen operaattori, eli yksioperandinen operaattori. Operaattori kohdistuu siis vain yhteen kohteeseen. Yhteenlaskuoperaattori (+) on binäärioperaattori, jossa on kaksi operandia. Kuinka yhteenlaskuoperaattori sitten ylikuormitetaan?

Tavoitteena on mahdollisuus esitellä kaksi Counter-muuttujaa ja laskea ne yhteen, kuten seuraavassa esimerkissä:

```
Counter varOne, varTwo, varThree;
varThree = varOne + varTwo;
```

Voisimme aloittaa nytkin kirjoittamalla funktion Add(), joka ottaa parametrikseen Counter-olion, laskee yhteen arvot ja palauttaa muutetun Counter-olion. Listaus 14.4. havainnollistaa tätä lähestymistapaa.

Listaus 14.4. Add()-funktio.

```
1:  // Listaus 14.4
2:  // Add-metodi
3:
4:  typedef unsigned short  USHORT;
5:
6:  #include <iostream.h>
7:
8:  class Counter
9:  {
10: public:
11:   Counter();
12:   Counter(USHORT initialValue);
13:   ~Counter(){}
14:   USHORT GetItsVal()const { return itsVal; }
15:   void SetItsVal(USHORT x) {itsVal = x; }
16:   Counter Add(const Counter &);
17:
18: private:
19:   USHORT itsVal;
20:
21: };
22:
23: Counter::Counter(USHORT initialValue):
24: itsVal(initialValue)
```

```
25: {}
26:
27: Counter::Counter():
28: itsVal(0)
29: {}
30:
31: Counter Counter::Add(const Counter & rhs)
32: {
33:   return Counter(itsVal+ rhs.GetItsVal());
34: }
35:
36: int main()
37: {
38:   Counter varOne(2), varTwo(4), varThree;
39:   varThree = varOne.Add(varTwo);
40:   cout << "varOne: " << varOne.GetItsVal()<< endl;
41:   cout << "varTwo: " << varTwo.GetItsVal() << endl;
42:   cout << "varThree: " << varThree.GetItsVal() << endl;
43:
44:   return 0;
45: }
```

Tulostus

```
varOne: 2
varTwo: 4
varThree: 6
```

Analyysi

Add()-funktio esitellään rivillä 16. Se ottaa parametrikseen vakion Counter-viittauksen, joka on nykyiseen olioon lisättävä luku. Se palauttaa Counter-olion, joka sijoitetaan rivillä 38. varOne on olio, varTwo on Add()-funktion parametri ja tulos sijoitetaan varThree-olioon.

Jotta voitaisiin luoda varThree alustamatta sen arvoa, tarvitaan oletusmuodostin. Oletusmuodostin alustaa itsVal-muuttujan arvoksi 0 (rivit 27-29). Koska varOne ja varTwo on alustettava nollasta poikkeavin arvoin, luodaan toinen muodostin (rivit 23-25). Toinen ratkaisu olisi antaa oletusarvo 0 rivillä 11 esitellylle muodostimelle.

operator+ ylikuormittaminen

Itse Add()-funktio on riveillä 30-33. Se toimii, mutta sen käyttö on hieman luonnotonta. Yhteenlaskuoperaattorin ylikuormittaminen takaisi Counter-luokan luonnollisemman käytön. Listaus 14.5 havainnollistaa tätä.

Listaus 14.5. Yhteenlaskuoperaattori.

```
1:  //operator+ ylimäärittely
2:
3:  typedef unsigned long  ULONG;
4:  #include <iostream.h>
5:
6:
7:  class Counter
```

```
8:  {
9:  public:
10:   Counter();
11:   Counter(ULONG initialValue);
12:   ~Counter(){}
13:   ULONG GetItsVal()const { return itsVal; }
14:   void SetItsVal(ULONG x) {itsVal = x; }
15:   Counter operator+ (const Counter &);
16: private:
17:   ULONG itsVal;
18: };
19:
20: Counter::Counter(ULONG initialValue):
21: itsVal(initialValue)
22: {}
23:
24: Counter::Counter():
25: itsVal(0)
26: {}
27:
28: Counter Counter::operator+ (const Counter & rhs)
29: {
30:   return Counter(itsVal + rhs.GetItsVal());
31: }
32:
33: int main()
34: {
35:   Counter varOne(2), varTwo(4), varThree;
36:   varThree = varOne + varTwo;
37:   cout << "varOne: " << varOne.GetItsVal()<< endl;
38:   cout << "varTwo: " << varTwo.GetItsVal() << endl;
39:   cout << "varThree: " << varThree.GetItsVal() << endl;
40:
41:   return 0;
42: }
```

Tulostus

```
varOne: 2
varTwo: 4
varThree: 6
```

Analyysi

operator+ esitellään rivillä 15 ja määritellään riveillä 28-31. Vertaa näitä rivejä Add()-funktion riveihin; ne ovat lähes identtiset. Niiden käyttötavat ovat kuitenkin erilaiset. On luonnollisempaa koodata näin:

```
varThree = varOne + varTwo;
```

kuin näin:

```
varThree = varOne.Add(varTwo);
```

Ei kovinkaan suuri muutos, mutta tarpeeksi suuri tekemään ohjelmasta helpomman käyttää ja ymmärtää.

Operaattorin ylikuormittamisen rajoitukset

Sisäisten tietotyyppien (kuten int) operaattoreita ei voida ylikuormittaa. Niiden suoritusjärjestystä ei voida muuttaa eikä myöskään sitä, kuinka moneen operandiin (yhteen tai useampaan) ne kohdistuvat. Uusia operaattoreita ei voida luoda eli et voi luoda esimerkiksi operaattoria ** olemaan 'potenssiin korotus' -operaattori.

Mitä voidaan ylikuormittaa

Operaattorin ylikuormittaminen on hämärimpiä C++ -alueita uusille ohjelmoijille. On houkuttelevaa luoda uusia käyttötarkoituksia joillekin operaattoreille, mutta liika monimutkaisuus tekee koodista vaikealukuista.

Tietenkin yhteenlaskuoperaattorin muuttaminen vähennyslaskuksi tai kertolaskuoperaattorin yhteenlaskuksi voi olla hauskaa, mutta sellainen ei sovi ammattilaisohjelmoijille. Vaarat piilevät hyvää tarkoittavassa, mutta turhassa operaattorin käytössä; esimerkiksi + voisi tarkoittaa kirjainten yhdistämistä ja / taas merkkijonon pilkkomista. Tuollaista käyttöä voisi hyvinkin harkita, mutta tulee muistaa, että ylikuormittamisen tarkoituksena on käytettävyyden ja ymmärrettävyyden lisääminen.

operator=

Kääntäjä antaa siis käyttöön oletusmuodostimen, tuhoajafunktion ja kopiomuodostimen. Neljäs ja viimeinen kääntäjän tarjoama toiminto (ellet määritä uusia) on sijoitusoperaattori (operator=()).

Tätä operaattoria kutsutaan aina, kun sijoitat olioon. Esimerkiksi:

```
CAT catOne(5,7);
CAT catTwo(3,4);
// muuta koodia…
catTwo = catOne;
```

Aluksi luodaan oliot catOne ja catTwo. Ne alustetaan alkuarvoin.

Huomaa, että tässä tapauksessa ei kutsuta kopiomuodostinta. catTwo on jo olemassa; sitä ei tarvitse muodostaa.

Luvussa 13, "Kehittyneet funktiot", keskusteltiin pinnallisten ja jäsenkohtaisten kopioiden sekä tarkkojen kopioiden eroista. Pinnallinen kopio kopioi vain jäsenet, jolloin molemmat oliot päätyvät osoittamaan samaan muistialueeseen. Tarkka kopio varaa tarvittavan muistin. Kuva 13.1 havainnollisti tätä ajattelua; tutki kuvaa uudelleen virkistääksesi muistiasi.

Sama ajattelutapa koskee sijoitusta kuin kopiomuodostinta. Sijoitusoperaattoria koskee kuitenkin vielä yksi lisäseikka. Olio catTwo on jo

olemassa ja muisti on jo varattu. Tuo muisti täytyy vapauttaa haluttaessa välttää muistikato.

Niinpä ensimmäinen seikka sijoitusoperaattoria toteutettaessa on tuhota muisti, johon osoittimet osoittavat. Mutta mitä tapahtuu, jos sijoitat catTwo-olion itseensä seuraavalla tavalla:

```
catTwo = catTwo;
```

Kukaan ei tietenkään tee noin tarkoituksella, mutta ohjelman on kyettävä selviytymään siitä. Ja vielä tärkeämpää, tällaista voi sattua vahingossa, kun viitatut ja uudelleenviitatut osoittimet kätkevät sen seikan, että olion sijoitus tapahtuu siihen itseensä.

Jos et selvittäisi tuota ongelmaa huolellisesti, catTwo tuhoaisi oman muistivarauksensa. Sen jälkeen, kun se olisi valmis kopioimaan muistiin kohteen oikealta puolelta sijoitusoperaattoria, syntyisi hyvin suuri ongelma; muistia ei enää olisikaan.

Suojautuakseen tältä sijoitusoperaattorin täytyy tarkistaa, onko sijoitusoperaattorin oikealla puolella olio itse. Se tekee sen tutkimalla this-osoitinta. Listaus 14.6 esittelee luokan, jossa on sijoitusoperaattori.

Listaus 14.6. Sijoitusoperaattori.

```
1:  // Kopiomuodostimet
2:
3:  #include <iostream.h>
4:
5:  class CAT
6:  {
7:  public:
8:    CAT();                      // oletusmuodostin
9:    // Kopiomuodostin ja tuhoaja ei mukana
10:   int GetAge() const { return *itsAge; }
11:   int GetWeight() const { return *itsWeight; }
12:   void SetAge(int age) { *itsAge = age; }
13:   CAT operator=(const CAT &);
14:
15: private:
16:   int *itsAge;
17:   int *itsWeight;
18: };
19:
20: CAT::CAT()
21: {
22:   itsAge = new int;
23:   itsWeight = new int;
24:   *itsAge = 5;
25:   *itsWeight = 9;
26: }
27:
28:
29: CAT CAT::operator=(const CAT & rhs)
30: {
31:   if (this == &rhs)    // tarkistetaan, ovatko oliot samoja
```

```
32:    return *this;
33:    delete itsAge;
34:    delete itsWeight;
35:    itsAge = new int;
36:    itsWeight = new int;
37:    *itsAge = rhs.GetAge();
38:    *itsWeight = rhs.GetWeight();
39:    return *this;
40: }
41:
42:
43: int main()
44: {
45:    CAT frisky;
46:    cout << "frisky's age: " << frisky.GetAge() << endl;
47:    cout << "Setting frisky to 6...\n";
48:    frisky.SetAge(6);
49:    CAT whiskers;
50:    cout << "whiskers' age: " << whiskers.GetAge() << endl;
51:    cout << "copying frisky to whiskers...\n";
52:    whiskers = frisky;
53:    cout << "whiskers' age: " << whiskers.GetAge() << endl;
54:    return 0;
55: }
```

Tulostus

```
frisky's age: 5
Setting frisky to 6
whiskers' age: 5
copying frisky to whiskers…
whiskers' age: 6
```

Analyysi

Listaus 14.6 tuo takaisin CAT-luokan ja siinä ei enää tilan säästämiseksi ole kopiomuodostinta eikä tuhoajafunktiota. Rivillä 14 esitellään sijoitusoperaattori, joka määritellään sitten riveillä 29-37.

Rivillä 32 testataan nykyinen olio (CAT, johon sijoitetaan), jotta nähtäisiin, onko se sama kuin CAT, joka sijoitetaan. Tämä toteutetaan tarkistamalla, onko rhs:n osoite sama kuin this-osoittimessa oleva osoite. Vaihtoehtoisesti voidaan testaus suorittaa viittaamalla this-osoittimella uudelleen ja katsomalla ovatko oliot samat:

```
if (*this == rhs)
```

Tietenkin myös yhtäsuuruusoperaattori (==) voidaan ylikuormittaa, jolloin voidaan määrätä, mitä olioiden yhtäsuuruus tarkoittaa.

Muunnosoperaattorit

Mitä tapahtuu yritettäessä sijoittaa sisäistä tyyppiä (int, unsigned short, tms.) oleva muuttuja käyttäjän määrittelemän luokan olioon? Listaus 14.7 palauttaa mieliin Counter-luokan ja yrittää sijoittaa Counter-olion int-muuttujaan.

Varoitus!

Listausta 14.7 ei voida kääntää!

Listaus 14.7. Counter-olion sijoittaminen int-muuttujaan.

```
1:  // Koodia ei voida kääntää
2:
3:  typedef unsigned short  USHORT;
4:
5:  #include <iostream.h>
6:
7:  class Counter
8:  {
9:  public:
10:   Counter();
11:   ~Counter(){}
12:   USHORT GetItsVal()const { return itsVal; }
13:   void SetItsVal(USHORT x) {itsVal = x; }
14: private:
15:   USHORT itsVal;
16:
17: };
18:
19: Counter::Counter():
20: itsVal(0)
21: {}
22:
23: int main()
24: {
25:   USHORT theShort = 5;
26:   Counter theCtr = theShort;
27:   cout << "theCtr: " << theCtr.GetItsVal() << endl;
28:   return ;0
29: }
```

Tulostus

```
Compiler error! Unable to convert int to Counter
```

Analyysi

Riveillä 7-17 esitellyllä Counter-luokalla on vain oletusmuodostin. Siinä ei ole mitään erityistä metodia int-muuttujan muuntamiseksi Counter-olioksi, joten rivi 26 aiheuttaa kääntäjän virheilmoituksen. Kääntäjä ei osaa päätellä, ellet kerro sitä, että annettu int tulisi sijoittaa olion jäsenmuuttujaan itsVal.

Listaus 14.8 korjaa tilanteen luomalla muunnosoperaattorin: muodostimen, joka ottaa parametrikseen int-muuttujan ja tuottaa Counter-olion.

Listaus 14.8. int-muunnos Counter-olioksi.

```
1:  // Listaus 14.12
2:  // Muodostin muunnosoperaattorina
3:
4:  typedef unsigned short  USHORT;
5:
6:  #include <iostream.h>
7:
8:  class Counter
9:  {
10: public:
11:   Counter();
12:   Counter(USHORT val);
13:   ~Counter(){}
14:   USHORT GetItsVal()const { return itsVal; }
15:   void SetItsVal(USHORT x) {itsVal = x; }
16: private:
17:   USHORT itsVal;
18:
19: };
20:
21: Counter::Counter():
22: itsVal(0)
23: {}
24:
25: Counter::Counter(USHORT val):
26: itsVal(val)
27: {}
28:
29:
30: int main()
31: {
32:   USHORT theShort = 5;
33:   Counter theCtr = theShort;
34:   cout << "theCtr: " << theCtr.GetItsVal() << endl;
35:   return 0;
36: }
```

Tulostus

```
theCtr: 5
```

Analyysi

Rivillä 12 on tuo tärkeä muutos. Siinä muodostin ylikuormitetaan ottamaan parametrikseen int-tyypin. Muodostin toteutetaan riveillä 25-27. Muodostin luo Counter-olion, jossa int on mukana.

Nyt kääntäjä pystyy kutsumaan muodostinta, joka ottaa int-argumentin. Katso kuitenkin, mitä tapahtuu, jos yrität kääntää sijoituksen seuraavasti:

```
1:  Counter theCtr(5);
2:  USHORT theShort = theCtr;
3:  cout << "theShort : " << theShort  << endl;
```

Koodi aiheuttaisi taasen kääntäjän virheen. Vaikka kääntäjä tietääkin nyt, kuinka Counter-olio muodostetaan int-tyypistä, se ei osaa tehdä käänteistä operaatiota.

Operaattori unsigned short()

Ratkaistakseen edellä kuvatun ja muut samanlaiset ongelmat C++ sisältää muunnosoperaattoreita, joita voidaan lisätä luokkaan. Luokka voi määrittää, kuinka tehdä muunnokset sisäisiin tyyppeihin. Listaus 14.9 havainnollistaa tätä. Yksi huomautus kuitenkin: muunnosoperaattorit eivät määritä palautusarvoa, vaikka ne palauttavatkin muunnetun arvon.

Listaus 14.9. Counter-olion muuntaminen unsigned short() -tyypiksi.

```
1:  // Listaus 14.9
2:  // Muodostin muuntajana
3:  typedef unsigned short  USHORT;
4:
5:  #include <iostream.h>
6:
7:  class Counter
8:  {
9:  public:
10:   Counter();
11:   Counter(USHORT val);
12:   ~Counter(){}
13:   USHORT GetItsVal()const { return itsVal; }
14:   void SetItsVal(USHORT x) {itsVal = x; }
15:   operator unsigned short();
16: private:
17:   USHORT itsVal;
18:
19: };
20:
21: Counter::Counter():
22: itsVal(0)
23: {}
24:
25: Counter::Counter(USHORT val):
26: itsVal(val)
27: {}
28:
29: Counter::operator unsigned short()
30: {
31:   return (int (itsval) );
32: }
33:
34: int main()
35: {
36:   Counter ctr(5);
37:   int theShort = ctr;
38:   cout << "theshort: " << theShort << endl;
39:   return 0;
40: }
```

Tulostus

```
theShort: 5
```

Analyysi
Rivillä 15 esitellään muunnosperaattori. Huomaa, että sillä ei ole palautusarvoa. Funktion toteutus on riveillä 29-32. Rivi 31 palauttaa itsVal-arvon muunnettuna int-tyypiksi.

Nyt kääntäjä tietää, kuinka int-tyyppejä muunnetaan Counter-olioiksi ja päinvastoin ja ne voidaan sijoittaa vapaasti toisiinsa.

Yhteenveto

Tässä luvussa käsiteltiin operaattorin ylikuormittamista.

Kääntäjä tarjoaa kopiomuodostimen ja operator= -operaattorin, ellet luo omia. Ne tekevät kuitenkin vain jäsenkohtaisen kopion luokasta. Sellaisten luokkien kohdalla, joissa on jäsentietoina osoittimia vapaaseen muistiin, täytyy nuo metodit korvata niin, että muistia varataan kohdeoliolle.

Melkein kaikki C++ -operaattorit voidaan ylikuormittaa, vaikkakin turhanpäiväistä ylikuormittamista tulee välttää. Operaattoreiden operandimäärää ei voida muuttaa eikä myöskään kehittää uusia operaattoreita.

this-osoitin viittaa nykyiseen olioon ja se on kaikkien jäsenfunktioiden näkymätön parametri. Ylikuormitetut operaattorit palauttavat usein uudelleenviitatun this-osoittimen.

Muunnosoperaattoreilla voidaan luoda luokkia, joita voidaan käyttää ilmauksissa, joissa yleensä käytetään eri tyyppisiä kohteita. Ne ovat poikkeuksia sääntöön, jonka mukaan kaikki funktiot palauttavat ulkoisen arvon. Muodostimien ja tuhoajafunktioiden tapaan niillä ei ole palautusarvoa.

Kysymyksiä ja Vastauksia

K
Miksi ylikuormittaisin operaattorin, kun voin helposti luoda metodin?
V
On helpompi käyttää ylikuormitettuja operaattoreita silloin, kun niiden käyttäytyminen on selkeää. Ylikuormittamisella voidaan matkia sisäisten tyyppien toiminnallisuutta.

K
Mitä eroa on kopiomuodostimella ja sijoitusoperaattorilla?
V
Kopiomuodostin luo uuden olion, jolla on samat arvot kuin kohdeoliolla. Sijoitusoperaattori muuttaa olemassa olevaa oliota niin, että sillä on samat arvot kuin toisella oliolla.

K
Mitä tapahtuu jälkiliiteoperaattorissa käytetylle int-tyypille?
V
Ei mitään. Tuota int-tyyppiä ei koskaan käytetä muutoin kuin lippuna etuliite- ja jälkiliiteoperaattoreiden ylikuormituksessa.