Muodostinfunktiot, tuhoajafunktiot ja sijoitusoperaattorit

Melkein jokaisella kirjoittamallasi luokalla on yksi tai useampi muodostinfunktio, tuhoajafunktio ja sijoitusoperaattori. Tämä on pieni ihme. Nämä funktiot ovat voi leipäsi päällä eli ne funktiot, jotka hallitsevat perustoimintoja tuomalla uuden olion esiin ja varmistamalla, että se on alustettu, hankkiutumalla eroon oliosta ja varmistamalla, että se on kunnolla saneerattu, ja antamalla oliolle uuden arvon. Virheiden tekeminen näissä funktioissa voi johtaa kaikkien luokkiesi kohdalla kauaskantoisiin ja erityisen epämukaviin jälkimaininkeihin, joten on elintärkeää, että teet ne oikein. Tarjoan tässä kappaleessa opastusta niiden funktioiden kokoamisessa, jotka muodostavat hyvinmuodostettujen luokkien selkärangan.

Kohta 11: Esittele kopiomuodostin ja sijoitusoperaattori luokille, joilla on dynaamisesti varattua muistia.

Tutki luokkaa, joka edustaa `String`-olioita:

```
// huonosti suunniteltu String-luokka
class String {
public:
  String(const char *value);
  ~String();

  ...                          // ei kopiom. tai operator=

private:
  char *data;
};
```

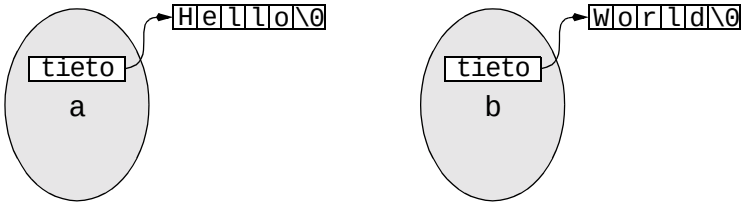
```
String::String(const char *value)
{
  if (value) {
    data = new char[strlen(value) + 1];
    strcpy(data, value);
  }
  else {
    data = new char[1];
    *data = '\0';
  }
}

inline String::~String() { delete [] data; }
```

Huomaa, että tässä luokassa ei ole esiteltynä sijoitusoperaattoria tai kopiomuodostinta. Kuten näet, tästä on eräitä ikäviä seurauksia.

Jos teet nämä oliomääritykset,

```
String a("Hello");
String b("World");
```

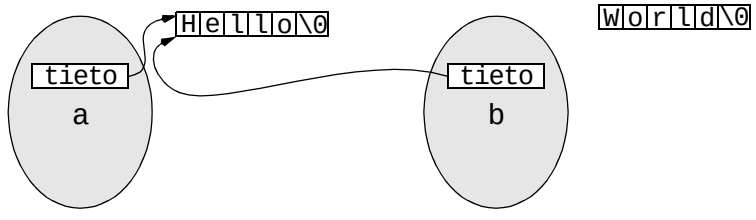
tilanne nähdään alla:



Olion `a` sisällä on osoitin muistiin, joka sisältää merkkijonon "Hello". Tästä on erillään olio `b`, joka sisältää osoittimen merkkijonoon "World". Jos suoritat nyt sijoituksen

```
b = a;
```

ei ole olemassa mitään asiakkaan määrittelemää kutsuttavaa `operator=`-funktiota, joten C++-kieli luo ja kutsuu sen sijaan oletuksena olevaa sijoitusoperaattoria (katso Kohta 45). Tämä suorittaa jäsentason sijoituksen `a`-olion jäsenistä `b`-olion jäseniin, mikä osoittimien kannalta (`a.data` ja `b.data`) on vain bittitason kopiointi. Tämän sijoituksen tulos nähdään alla.



Tässä tilanteessa on ainakin kaksi ongelmaa. Ensinnäkään muistia, johon `b`-olio viittaa, ei koskaan poistettu; se menetettiin lopullisesti. Tämä on klassinen esimerkki siitä, kuinka muistivuoto saadaan aikaan. Toiseksi sekä `a` että `b` sisältävät nyt osoittimet samaan merkkijonoon. Kun jompikumpi niistä menee näkyvyysalueensa ulkopuolelle, sen tuhoajafunktio poistaa muistin, johon toinen vielä osoittaa. Esimerkiksi:

```
String a("Hello");              // määritä ja muodosta a
{                               // avaa uusi näkyvyysalue
  String b("World");            // määritä ja muodosta b
  ...
  b = a;                        // suorita oletus op=,
                                // menetä b:n muisti
}                               // sulje näkyvyysalue,
                                // kutsu b:n
                                // tuhoajafunktiota
String c = a;                   // c.data on
                                // määrittelemätön!
                                // a.data on jo poistettu
```

Tämän esimerkin viimeinen lause on kutsu kopiomuodostimeen, jota ei myöskään ole määritetty luokassa. Siitä johtuu, että C++ tulee luomaan sen samaan tapaan kuin sijoitusoperaattorin (katso jälleen Kohta 45) ja se saa aikaan saman käyttäytymisen: perustana olevien osoittimien bittitason kopioinnin. Tämä johtaa saman kaltaiseen ongelmaan, mutta nyt ei tarvitse murehtia muistivuotoa, koska alustuksen kohteena oleva olio ei vielä voi osoittaa mihinkään varattuun muistiin. Esimerkiksi: Muistivuotoa ei tapahdu yllä olevan lähdetekstin tapauksessa silloin, kun `c.data` alustetaan `a.data`:n arvolla, koska `c.data` ei vielä osoita minnekään. Kuitenkin sen jälkeen, kun `c` on alustettu `a`:n arvolla, sekä `c.data` että `a.data` osoittavat samaan paikkaan, joten se paikka tullaan poistamaan kahdesti: kerran silloin, kun `c` tuhotaan ja uudestaan silloin, kun `a` tuhotaan.

Kopiointimuodostimen tapaus eroaa kuitenkin vähän sijoitusoperaattorin tapauksesta tavalla, joka voi vaikuttaa sinuun: arvoparametrillä välittäminen. Kohdassa 21 esitetään tietysti, että olioita pitäisi välittää vain harvoin arvoparametrillä, mutta tutki kuitenkin tätä:

```
void doNothing(String localString) {}
String s = "The Truth Is Out There";
doNothing(s);
```

Kaikki näyttää ihan viattomalta, mutta koska `localString` välitetään arvoparametrillä, se täytyy alustaa `s`:n arvolla oletuksena olevan kopiomuodostimen kautta. Tästä johtuu, että `localString`-muuttujalla on kopio *osoittimesta*, joka on `s`:n sisällä. Kun `doNothing` lopettaa suorituksensa, `localString` menee

näkyvyysalueensa ulkopuolelle ja sen tuhoajafunktiota kutsutaan. Lopputulos on jo varmaan tuttu: `s` sisältää osoittimen muistiin, jonka `localString` on jo poistanut.

Jos `delete`-funktiota käytetään siinä osoittimessa, joka on jo poistettu, tulos on muuten määrittelemätön, joten vaikka `s`-muuttujaa ei ikinä enää käytettäisi, asiasta saattaisi silti aiheutua ongelma, kun se menee näkyvyysalueensa ulkopuolelle.

Ratkaisu tämän kaltaisiin ongelmiin peitenimen käytössä osoittimessa on kirjoittaa omat versiosi kopiomuodostimesta ja sijoitusoperaattorista, jos sinulla on osoittimia luokassasi. Joko voidaan kopioida osoituksen kohteena olleet tietorakenteet näiden funktioiden sisällä niin, että jokaisella oliolla on niistä oma kopio, tai voidaan toteuttaa eräänlainen viittaukset laskeva skeema, jolla pidetään kirjaa siitä, kuinka monta oliota osoittaa määrättyyn tietorakenteeseen. Työtapa, joka laskee viittaukset, on enemmän kuin monimutkainen, ja vaatii myös ylimääräistä työtä muodostin- ja tuhoajafunktioissa, mutta joissakin (ei toki kaikissa) sovelluksissa tästä voi olla tuloksena huomattavaa muistin säästöä ja vankka lisäys nopeuteen.

Eräille luokille tuottaa enemmän vaikeuksia kuin on kannattavaa toteuttaa kopiomuodostimia ja sijoitusoperaattoreita, varsinkin silloin, kun sinulla on syytä uskoa, että asiakkaasi eivät kopioi tai suorita sijoituksia. Edelliset esimerkit osoittavat, että vastaavien jäsenfunktioiden mainitsematta jättäminen heijastaa heikkoa suunnittelua, mutta minkä voit, jos niiden kirjoittaminen ei myöskään ole käytännöllistä? Yksinkertaista: noudatat tämän Kohdan ohjeita. *Esittelet* funktiot (ilmenee, että ne ovat tyypiltään `private`), mutta et määritä (eli toteuta) niitä ollenkaan. Tämä estää asiakkaita kutsumasta niitä ja myös kääntäjiä luomasta niitä. Jos haluat lukea yksityiskohtia tästä siististä tempusta, katso Kohta 27.

Vielä yksi asia tässä Kohdassa käyttämästäni `String`-luokasta. Olin huolellinen muodostinfunktion runko-osassa ja käytin hakasulkeita (`[]`) `new`-operaattorin kanssa molemmilla kerroilla, kun kutsuin sitä, vaikka yhdessä kohdassa halusin vain yhden yksittäisen olion. Kuten kuvasin Kohdassa 5, on elintärkeää työllistää samat muodot vastaavissa ohjelmissa `new`- ja `delete`-operaattoreista, joten olin huolellisesti yhdenmukainen, kun käytin `new`-operaattoria. Tämä on asia, jota sinun ei pidä unohtaa. Varmista *aina*, että käytät hakasulkeita (`[]`) `delete`-operaattorin kanssa, jos ja vain jos käytit hakasulkeita myös `new`-operaattorissa.

Kohta 12: Suosi alustamista sijoituksen sijasta muodostinfunktioissa.

Tutki mallia, joka on tarkoitettu niiden luokkien luontiin, jotka sallivat nimen liittämisen `T`-tyyppisen olion osoittimeen:

```
template<class T>
class NamedPtr {
public:
  NamedPtr(const string& initName, T *initPtr);
  ...
private:
  string name;
  T *ptr;
};
```

(Kun otetaan huomioon peitenimien käyttö, joka saattaa ilmetä osoitinjäsenillä varustettujen olioiden sijoittamisen ja kopiomuodostamisen aikana (katso Kohta 11), haluat varmaan harkita sitä, pitääkö `NamedPtr`-luokan toteuttaa nämä funktiot. Vihje: sen pitäisi (katso Kohta 27).)

Kun kirjoitat `NamedPtr`-muodostinfunktion, sinun täytyy siirtää näiden parametrien arvot vastaaviin tietojäseniin. On olemassa kaksi tapaa tehdä tämä. Ensimmäinen on käyttää jäsenten alustuslistaa:

```
template<class T>
NamedPtr<T>::NamedPtr(const string& initName, T *initPtr)
: name(initName), ptr(initPtr)
{}
```

Toinen tapa on tehdä sijoitukset muodostinfunktion rungossa:

```
template<class T>
NamedPtr<T>::NamedPtr(const string& initName, T *initPtr)
{
  name = initName;
  ptr = initPtr;
}
```

Näiden näkemysten välillä on huomattavia eroja.

Pelkästään pragmaattisen näkemyksen mukaan on aikoja, jolloin alustuslistaa *täytyy* käyttää. Erityisesti `const`-määre ja viittausjäsenet voidaan *pelkästään* alustaa, niitä ei koskaan sijoiteta. Joten jos olet päättänyt, että `NamedPtr<T>`-olio ei voi koskaan muuttaa nimeään tai osoitintaan, sinun kannattaa noudattaa Kohdan 21 ohjetta ja esitellä jäsenet `const`-tyyppisinä:

```
template<class T>
class NamedPtr {
public:
  NamedPtr(const string& initName, T *initPtr);
  ...
private:
  const string name;
  T * const ptr;
};
```

Tämä luokkamääritys *vaatii* sen, että käytät jäsenten alustuslistaa, koska `const`-tyyppiset jäsenet voidaan vain alustaa, ei koskaan sijoittaa.

Olisit saanut haltuusi aivan erilaisen käyttäytymisen, jos olisit päättänyt, että `NamedPtr<T>`-olion pitäisi sisältää *viittaus* olemassaolevaan nimeen. Sinun olisi myös silloin pitänyt alustaa viittaus muodostinfunktion jäsenten alustuslistoilla. Voisit myös tietysti yhdistää nämä kaksi tuottamalla `NamedPtr<T>`-olioita vain luku-oikeudella varustettuihin nimiin, joita voitaisiin muokata luokan ulkopuolella:

```
template<class T>
class NamedPtr {
public:
  NamedPtr(const string& initName, T *initPtr);
  ...

private:
  const string& name;          // täytyy tulla alustetuksi
                               // alustuslistan kautta

  T * const ptr;               // täytyy tulla alustetuksi
                               // alustuslistan kautta
};
```

Alkuperäinen luokkamalli ei kuitenkaan sisällä `const`-määrettä tai viittausjäseniä. Jäsenten alustuslistan käyttäminen on silti suositeltavaa silloin, kun muodostinfunktion sisäpuolelle suoritetaan sijoitustoimintoja. Tällä kertaa syy on tehokkuus. Kun käytetään jäsenten alustuslistaa, kutsutaan vain yhtä `string`-tyyppistä jäsentä. Silloin, kun muodostinfunktion sisällä tapahtuvaa sijoitusta käytetään, kutsutaan kahta jäsentä. Jotta ymmärtäisit miksi, tutki mitä tapahtuu, kun esittelet olion `NamedPtr<T>`.

Olioiden muodostus tapahtuu kahdessa vaiheessa:

1. Tietojäsenten alustus. (Katso myös Kohta 13.)
2. Kutsutun muodostinfunktion rungon suoritus.

(Olioille, joilla on kantaluokat, kantaluokan jäsenen alustus ja muodostinfunktion rungon suoritus tapahtuu aikaisemmin kuin periytyneiden luokkien.)

Tämä tarkoittaa `NamedPtr`-luokille sitä, että `string`-tyyppisen olion `name`-muodostinfunktiota kutsutaan *aina* ennen kuin pääset `NamedPtr`-muodostinfunktion rungon sisälle. Ainoa kysymys tällöin kuuluu: mitä `string`-muodostinfunktiota kutsutaan?

Tämä riippuu `NamedPtr`-luokkien jäsenten alustuslistasta. Jos et määritä alustusargumenttia `name`-oliolle, oletuksena olevaa `string`-tyyppistä muodostinfunktiota kutsutaan. Kun myöhemmin suoritat sijoituksen `NamedPtr`-muodostinfunktion sisällä olevaan `name`-olioon, kutsut `operator=`-funktiota `name`-olioon. Tästä on tuloksena kaksi kutsua `string`-tyyppisiin jäsenfunktioihin: yksi oletusmuodostinfunktioon ja toinen sijoitukseen.

Jos toisaalta käytät jäsenten alustuslistaa määrittääksesi, että `name`-olio alustetaan `initName`-arvolla, `name` tullaan alustamaan kopiomuodostimella yhden yksittäisen funktiokutsun kustannuksella.

Tarpeettoman funktiokutsun kustannukset voivat olla merkittävät myös alhaisen `string`-tyyppisen muuttujan kohdalla, ja kun luokat tulevat isommiksi ja monimutkaisemmiksi, niin tapahtuu myös muodostinfunktioille ja olioiden muodostamisen kustannuksille. Jos vakiinnutat sen tavan, että käytät jäsenen alustuslistaa aina kun voit, et ainoastaan tyydytä `const`-määreen ja viittausjäsenten vaatimuksen tarvetta, vaan minimoit myös mahdollisuudet, että tietojäsenet alustetaan tehottomalla tavalla.

Toisin sanoen, alustus jäsenten alustuslistan kautta on *aina* laillista, se ei ole *koskaan* tehottomampaa kuin sijoitus muodostinfunktion rungon sisällä, ja se on usein *tehokkaampaa*. Luokan ylläpito lisäksi yksinkertaistuu, koska jos tietojäsenen tyyppiä muutetaan myöhemmin jäsenten alustuslistan käyttöä vaativaksi, sinun ei tarvitse muuttaa mitään.

On kuitenkin yksi tapaus, jolloin on järkevää käyttää sijoitusta luokan tietojäsenten alustuksen sijasta. Se on silloin, kun on suuri joukko tyypiltään *sisäänrakennettuja* tietojäseniä, ja haluat, että ne kaikki alustetaan samalla tavalla jokaisessa muodostinfunktiossa. Tässä on esimerkiksi luokka, jolle olisi tarpeellista tämän kaltainen käsittely:

```
class ManyDataMbrs {
public:
  // oletusmuodostinfunktio
  ManyDataMbrs();

  // kopiomuodostin
  ManyDataMbrs(const ManyDataMbrs& x);

private:
  int a, b, c, d, e, f, g, h;
  double i, j, k, l, m;
};
```

Oletetaan, että haluat alustaa kaikki kokonaisluvut (int) arvoon 1 ja kaikki `double`-tyyppiset muuttujat arvoon 0 myös silloin, kun kopiomuodostinta käytetään. Jos käytät jäsenten alustuslistoja, sinun täytyy kirjoittaa tällä tavalla:

```
ManyDataMbrs::ManyDataMbrs()
: a(1), b(1), c(1), d(1), e(1), f(1), g(1), h(1), i(0),
  j(0), k(0), l(0), m(0)
{ ... }

ManyDataMbrs::ManyDataMbrs(const ManyDataMbrs& x)
: a(1), b(1), c(1), d(1), e(1), f(1), g(1), h(1), i(0),
  j(0), k(0), l(0), m(0)
{ ... }
```

Tämä on vieläkin epämiellyttävämpää kuin raataminen. Se on lyhyellä aikavälillä altis virheille ja pitkällä tähtäyksellä vaikeaa ylläpitää.

Voit kuitenkin hyötyä siitä, että alustuksen ja sisäänrakennetuista tyypeistä koostuvien (ei-`const`-, ei-viittaus-) olioiden sijoituksen välillä ei ole toiminnallista eroa, joten voit turvallisesti korvata jäsentyyppiset alustuslistat funktiokutsulla yleiseen alustusrutiiniin:

```
class ManyDataMbrs {
public:
  // oletusmuodostin
  ManyDataMbrs();

  // kopiomuodostin
  ManyDataMbrs(const ManyDataMbrs& x);

private:
  int a, b, c, d, e, f, g, h;
  double i, j, k, l, m;

  void init();                  // käytetään alustamaan
                                // tietojäsenet
};

void ManyDataMbrs::init()
{
  a = b = c = d = e = f = g = h = 1;
  i = j = k = l = m = 0;
}

ManyDataMbrs::ManyDataMbrs()
{
  init();

  ...

}

ManyDataMbrs::ManyDataMbrs(const ManyDataMbrs& x)
{
  init();

  ...

}
```

Koska alustusrutiini on luokan toteutuksen yksityiskohta, olet tietenkin huolellinen ja teet siitä tyypiltään `private`, eikö niin?

Huomaa, että luokan `static`-tyyppisiä jäseniä ei *koskaan* pidä alustaa luokan muodostinfunktiossa. Staattiset jäsenet alustetaan vain kerran ohjelman suorituksen aikana, joten ei ole ollenkaan järkevää "alustaa" niitä joka kerta, kun luokan tyyppinen olio luodaan. Näin tekeminen olisi ainakin tehotonta: mitä hyödyttää "alustaa" olio monta kertaa? Luokan staattisten jäsenten alustaminen on sitä paitsi niin erilaista kuin niiden epästaattisten vastineiden alustaminen, että kokonainen Kohta - Kohta 47 - on omistettu tälle aiheelle.

Kohta 13: Luettele alustuslistan jäsenet esittelyjärjestyksessä.

Katumattomat Pascal- ja Ada-ohjelmoijat haikailevat usein kykyä määrittää taulukoita, joissa on mielivaltaiset rajat, esimerkiksi arvojen 0 - 10 sijasta 10 - 20. Työelämässä kauan olleet C-ohjelmoijat väittävät, että tosimiehet/-naiset aloittavat laskemisen aina nollasta, mutta `begin/end`-porukan lepyttäminen on onneksi helppoa. Sinun tarvitsee vain määrittää oma `Array`-luokkamalli:

```
template<class T>
class Array {
public:
  Array(int lowBound, int highBound);
  ...

private:
  vector<T> data;               // taulukon tieto tallen-
                                // nettu vector-olioon;
                                // katso Kohta 49, jos
                                // haluat tietoa vector-
                                // mallista

  size_t size;                  // elementtien # taulukossa

  int lBound, hBound;           // ala- ja yläraja
};

template<class T>
Array<T>::Array(int lowBound, int highBound)
: size(highBound - lowBound + 1),
  lBound(lowBound), hBound(highBound),
  data(size)
{}
```

Teollisuudessa vahvaksi todettu muodostinfunktio suorittaisi parametriensä järkevyyden testaamisen varmistaakseen, että `highBound`-muuttuja on arvoltaan ainakin yhtä suuri kuin `lowBound`-muuttuja, mutta tässä tapauksessa on vielä yksi ilkeämpi

virhe: vaikka taulukon raja-arvot sisältäisivät täydellisen hyvät arvot, sinulla ei auttamatta ole aavistustakaan siitä, kuinka monta elementtiä `data`-taulukko sisältää.

Kuulen sinun valittavan, että "Kuinka asiat voivat olla näin?" "Alustin `size`-muuttujan huolellisesti välittäen sille `vector`-muodostinfunktion!" Et valitettavasti tehnyt niin - yritit vain tehdä niin. Pelin säännöt ovat sellaiset, että luokan jäsenet alustetaan *siinä järjestyksessä, kuin on esitelty luokassa*; järjestyksellä, jossa ne on lueteltu jäsenen alustuslistassa, ei ole mitään merkitystä. `Array`-taulukkosi luomissa luokissa `data` alustetaan aina ensin, sen jälkeen `size`, `lBound` ja `hBound`. Aina.

Vaikka se voi tuntua perverssiltä, siihen on syynsä. Tutki tätä skenaariota:

```
class Wacko {
public:
  Wacko(const char *s): s1(s), s2(0) {}
  Wacko(const Wacko& rhs): s2(rhs.s1), s1(0) {}

private:
  string s1, s2;
};

Wacko w1 = "Hello world!";
Wacko w2 = w1;
```

Jos jäsenet olisi alustettu siinä järjestyksessä kuin ne näkyvät alustuslistassa, tietojäsenet `w1` ja `w2` olisi muodostettu eri järjestyksessä. Palauta mieleen, että olion jäsenten tuhoajafunktioita kutsutaan aina käänteisessä järjestyksessä kuin niiden muodostinfunktiot. Täten, jos yllä oleva sallittaisiin, kääntäjien pitäisi pitää kirjaa järjestyksestä, jossa jäsenet olisi alustettu *jokaiselle oliolle*, pelkästään varmistaakseen sen, että tuhoajafunktioita olisi kutsuttu oikeassa järjestyksessä. Se olisi kallis esitys. Jos haluat välttää näitä kustannuksia, muodostamisen ja tuhoamisen järjestys on sama annetun tyypin kaikille olioille, ja alustuslistassa olevien jäsenten järjestyksellä ei ole merkitystä.

Itse asiassa, jos haluat todella olla nirso, vain epästaattiset tietojäsenet alustetaan tämän säännön mukaisesti. Staattiset tietojäsenet toimivat samalla tavalla kuin globaalit ja nimiavaruuden oliot, joten ne alustetaan vain kerran; katso Kohta 47, jos haluat yksityiskohtia. Kantaluokan tietojäsenet alustetaan lisäksi ennen luokan periytettyjä tietojäseniä, joten jos käytät periytyvyyttä, sinun täytyy luetella kantaluokan alustajat jäsentesi alustuslistojen alkuosassa. (Jos käytät *moni*perintää, kantaluokkasi alustetaan siinä järjestyksessä, jossa *periytät* niistä; järjestys, jossa ne on lueteltu jäsenen alustuslistassa, jätetään jälleen huomiotta. Jos kuitenkin käytät moniperintää, sinulla on luultavasti paljon muita tärkeitä asioita hoidettavana. Jos et käytä moniperintää, lue Kohta 43. Siinä kerrotaan ehdotuksia, jotka koskevat moniperinnän hermostuttavia näkökulmia.

Pääasia on tämä: jos haluat ymmärtää, mitä todella tapahtuu silloin, kun oliosi alustetaan, varmista, että luettelet jäsenet alustuslistassa siinä järjestyksessä kuin ne on esitelty luokassa.

Kohta 14: Varmista, että kantaluokilla on virtuaaliset tuhoajafunktiot.

Joskus luokan on käytännöllistä pitää kirjaa siitä, kuinka monta saman tyyppistä oliota on olemassa. Suoraviivainen tapa tehdä tämä on luoda staattinen luokan jäsen, joka laskee oliot. Jäsen alustetaan arvolla 0, sitä lisätään luokan muodostinfunktioissa ja arvoa vähennetään luokan tuhoajafunktioissa.

Saattaisit kuvitella sotilassovelluksen, jossa vihollisen kohteita edustava luokka voisi näyttää tämän kaltaiselta:

```
class EnemyTarget {
public:
  EnemyTarget() { ++numTargets; }
  EnemyTarget(const EnemyTarget&) { ++numTargets; }
  ~EnemyTarget() { --numTargets; }

  static size_t numberOfTargets()
  { return numTargets; }

  virtual bool destroy();           // palauttaa EnemyTar-
                                    // get-olion tuhoamis-
                                    // yrityksen onnistu-
                                    // misen tuloksen

private:
  static size_t numTargets;         // oliolaskuri
};

// luokan static-tyypit täytyy määrittää luokan
// ulkopuolella
// alustus on oletuksena 0
size_t EnemyTarget::numTargets;
```

Tämän luokan avulla tuskin saat sopimusta puolustusministeriön kanssa, mutta se riittäköön meidän tarkoitukseemme tässä tapauksessa, joka vaatii huomattavasti vähemmän kuin puolustusministeriön tarpeet. Tai ainakin toivomme niin.

Olettakaamme, että vihollisen erityinen kohde on panssarivaunu, jonka mallinnat, kuinkas muuten (katso Kohta 35), `EnemyTargetin` julkisesti periytettynä luokkana. Koska olet kiinnostunut sekä panssarivaunujen että viholliskohteiden kokonaismäärästä, räväytät saman tempun periytetyn luokan kohdalla, kuin minkä teit kantaluokan kohdalla:

```
class EnemyTank: public EnemyTarget {
public:
  EnemyTank() { ++numTanks; }

  EnemyTank(const EnemyTank& rhs)
  : EnemyTarget(rhs)
  { ++numTanks; }

  ~EnemyTank() { --numTanks; }

  static size_t numberOfTanks()
  { return numTanks; }

  virtual bool destroy();

private:
  static size_t numTanks;        // tankkien oliolaskuri
};

size_t EnemyTank::numTargets;   // laskurin alustus
```

Olettakaamme lopulta, että luot jossakin kohtaa ohjelmassasi `EnemyTank`-olion dynaamisesti käyttäen `new`-operaattoria, ja hankkiudut siitä myöhemmin eroon `delete`-operaattorin avulla:

```
EnemyTarget *targetPtr = new EnemyTank;

...

delete targetPtr;
```

Kaikki tähän mennessä tehty tuntuu täysin juhlapyhältä. Molemmat luokat peruuttavat tuhoajafunktiossa sen, minkä tekivät muodostinfunktiossa, ja ohjelmassasi ei todellakaan ole mitään väärin. Käytit ohjelmassasi `delete`-operaattoria huolellisesti sen jälkeen, kun olit saanut olion, jonka olit loihtinut `new`-operaattorilla. Joku asia tuntuu kuitenkin aiheuttavan vaikeuksia. Ohjelmasi käyttäytyminen on *määrittelemätöntä* - sinulla ei ole mitään keinoa tietää, *mitä* tapahtuu.

C++-kielen standardi on harvinaisen selkeä tässä tapauksessa. Kun yrität poistaa periytetyn luokkaolion kantaluokan osoittimen kautta, ja kantaluokalla ei ole virtuaalista (*nonvirtual*) tuhoajafunktiota (`EnemyTarget`-oliolla taas on), tulokset ovat määrittelemättömät. Tämä tarkoittaa sitä, että kääntäjät voivat luoda lähdetekstiä, joka voi tehdä mitä tahansa: alustaa kiintolevysi uudelleen, lähettää ehdottelevaa sähköpostia pomollesi, faxata lähdetekstin kilpailijallesi tai mitä tahansa. (Suorituksen aikana tapahtuu usein, että periytetyn luokan tuhoajafunktiota ei koskaan kutsuta. Tämä tarkoittaisi tässä esimerkissä sitä, että `EnemyTanks`-olioiden laskuria ei oikaistaisi silloin, kun `targetPtr` poistetaan. Panssarivaunujen laskuri olisi täten väärin, mikä olisi aika häiritsevä tulevaisuudenkuva taistelijoille, jotka ovat riippuvaisia paikkaansapitävästä tiedosta taistelukentiltä.)

Jos haluat välttää tämän ongelman, sinun pitää pelkästään tehdä `EnemyTarget`-tuhoajafunktiosta *virtuaalinen*. Tuhoajafunktion esitteleminen virtuaaliseksi varmistaa oikein määritetyn käyttäytymisen, joka tekee täsmälleen sen mitä haluat: sekä `EnemyTank`-luokan että `EnemyTarget`-luokan tuhoajafunktioita kutsutaan, ennen kuin on vapautettu muisti, joka tallentaa olion.

`EnemyTarget`-luokka sisältää nyt virtuaalifunktion ja kantaluokat sisältävät yleensä virtuaalifunktion. Virtuaalifunktioiden tarkoitushan on sallia käyttäytymisen muokkaus periytetyissä luokissa (katso Kohta 36), joten melkein kaikki kantaluokat sisältävät virtuaalifunktioita.

Jos luokka *ei* sisällä yhtään virtuaalifunktiota, tämä on yleensä osoituksena siitä, että sitä ei ole tarkoituskaan käyttää kantaluokkana. Kun luokkaa ei ole tarkoitettu käytettäväksi kantaluokkana, tuhoajafunktion tekeminen virtuaaliseksi on yleensä huono ajatus. Tutki tätä esimerkkiä, joka perustuu ARM:ssä käytyyn keskusteluun (katso Kohta 50):

```
// luokka, jolla esitetään kaksiulotteisia pisteitä
class Point {
public:
  Point(short int xCoord, short int yCoord);
  ~Point();

private:
  short int x, y;
};
```

Jos `short int` -tyyppi varaa 16 bittiä, `Point`-olio voi mahtua 32-bitin rekisteriin. `Point`-olio voidaan lisäksi välittää 32-bittisenä funktioille, jotka on kirjoitettu muilla ohjelmointikielillä kuten C tai FORTRAN. Jos `Point`-olion tuhoajafunktiosta on tehty virtuaalinen, tilanne kuitenkin muuttuu.

Virtuaalifunktioiden toteutus vaatii sen, että oliot kuljettavat mukanaan lisätietoa, jota voidaan käyttää suorituksen aikana määrittelemään, mitä virtuaalifunktioita olion kohdalla tultaisiin kutsumaan. Useimmissa kääntäjissä tämä ylimääräinen tieto on muodoltaan osoitin, jota kutsutaan nimellä `vptr` ("virtual table pointer", virtuaalinen tauluosoitin). `vptr` osoittaa funktio-osoittimista koostuvaan taulukkoon, jota kutsutaan nimellä `vtbl` ("virtual table", virtuaalinen taulu); jokaisella luokalla, jolla on virtuaalifunktioita, on siihen liittyvä `vtbl`. Silloin, kun olio pyytää avuksi virtuaalifunktioita, itse kutsuttu funktio määritetään seuraamalla olion `vptr`-osoitinta `vtbl`-osoittimeen ja tarkistamalla vastaava funktio-osoitin `vtbl`-osoittimesta.

Yksityiskohdat siitä, kuinka virtuaalifunktiot toteutetaan, eivät ole tärkeitä. Tärkeää *on* se, että jos `Point`-luokka sisältää virtuaalifunktion, sen tyyppiset oliot *kaksinkertaistavat* kokonsa implisiittisesti kahdesta 16-bittisestä `short`-arvosta kahteen 16-bittiseen `short`-arvoon sekä 32-bittiseen `vptr`-osoittimeen! `Point`-oliot eivät enää mahdu 32-bittiseen rekisteriin. C++-kielen `Point`-oliot eivät lisäksi enää näytä rakenteeltaan samanlaisilta kuin mitä on esitelty muussa kielessä kuten C, koska nii-

den vieraan kielen vastineensa eivät sisällä `vptr`-osoitinta. Tästä on tuloksena se, että `Point`-olioita ei ole enää mahdollista välittää funktioille ja funktioilta, jotka on kirjoitettu muilla kielillä, paitsi jos ei eksplisiittisesti korvata `vptr`-osoitinta, joka itsekin on toteutuksen yksityiskohta eikä täten ole siirrettävissä.

Pääasia on se, että kaikkien tuhoajafunktioiden esitteleminen vastikkeettomasti virtuaalisiksi on aivan yhtä väärin, kuin jos niitä ei esitellä virtuaalisiksi koskaan. Monet ihmiset tekevät itse asiassa yhteenvedon tilanteesta tällä tavalla: esittele virtuaalituhoajafunktio luokassa, jos ja ainoastaan jos tuo luokka sisältää ainakin yhden virtuaalifunktion.

Tämä on hyvä sääntö, ja se toimii useimmiten, mutta valitettavasti on mahdollista joutua ei-virtuaalisen tuhoajafunktion ongelman puraisemaksi myös silloin, kun virtuaalifunktioita ei ole. Kohdassa 13 tutkitaan esimerkiksi luokkamallia, jolla toteutetaan taulukot, joissa asiakkaat määrittelevät raja-arvot. Oletetaan, että päätät kirjoittaa mallin periytetyille luokille, jotka esittävät nimettyjä taulukkoja, toisin sanoen luokkia, joissa jokaisella taulukolla on nimi:

```
template<class T>              // kantaluokan malli
class Array {                  // (Kohdasta 13)
public:
Array(int lowBound, int highBound);
  ~Array();

private:
  vector<T> data;
  size_t size;
  int lBound, hBound;
};

template<class T>
class NamedArray: public Array<T> {
public:
  NamedArray(int lowBound, int highBound, const string& name);
  ...

private:
  string arrayName;
};
```

Jos muutat jotenkin `NamedArray`-osoittimen `Array`-osoittimeksi jossakin kohtaa ohjelmassasi ja käytät sitten `delete`-operaattoria `Array`-osoittimessa, sinut ohjataan välittömästi määrittelemättömän käytöksen todellisuuteen:

```
NamedArray<int> *pna =
  new NamedArray<int>(10, 20, "Impending Doom");

Array<int> *pa;

...
```

```
pa = pna;           // NamedArray<int>* -> Array<int>*
...
delete pa;          // määrittelemätön! (Lisää tähän
                    // Yöjutun teema); käytännössä
                    // pa->arrayName johtaa usein muisti-
                    // vuotoon, koska *pa:n NamedArray-osaa
                    // ei koskaan tulla tuhoamaan
```

Tämä tilanne voi tulla esiin useammin kuin voisit kuvitella, koska on aika yleistä, että halutaan ottaa jokin olemassaoleva luokka, kuten `Array` tässä tapauksessa, ja johtaa siitä luokka, joka tekee kaikki samat asiat ja enemmänkin. `NamedArray` ei määrittele mitään `Array`:n käyttäytymistä uudelleen - se perii sen kaikki funktiot ilman muutoksia - se vain lisää muutaman lisäominaisuuden. Ei-virtuaalisen tuhoajafunktion ongelma ei kuitenkaan hellitä.

On lopulta mainitsemisen arvoista, että voi olla mukavaa esitellä puhtaat virtuaalituhoajafunktiot samoissa luokissa. Palauta mieleen, että puhtaista virtuaalifunktioista on seurauksena *abstraktit* luokat - joita ei voi instantioida (toisin sanoen et voi luoda sen tyyppisiä olioita). Sinulla voi kuitenkin joskus olla luokka, jonka haluaisit olevan abstrakti, mutta sinulla ei satu olemaan yhtään puhdasta virtuaalifunktiota. Mitä teet? Koska abstrakti luokka on tarkoitettu käytettäväksi kantaluokkana, ja koska kantaluokalla pitäisi olla virtuaalinen tuhoajafunktio, ja koska koska puhdas virtuaalifunktio saa aikaan abstraktin luokan, ratkaisu on yksinkertainen: esittele puhdas virtuaalinen tuhoajafunktio siinä luokassa, jonka haluat olevan abstrakti.

Tässä on esimerkki:

```
class AWOV {                    // AWOV = "Abstract w/o
                                // Virtuals"
public:
  virtual ~AWOV() = 0;          // esittele puhdas virtu-
                                // aalimuodostin
};
```

Tällä luokalla on puhdas virtuaalifunktio, joten se on abstrakti, ja sillä on virtuaalinen tuhoajafunktio, joten voit levätä huojentuneena, sillä sinun ei tarvitse huolehtia tuhoajafunktio-ongelmasta. On kuitenkin yksi ryppy: sinun täytyy lisätä puhtaan virtuaalituhoajafunktion *määritys*:

```
AWOV::~AWOV() {}                // puhtaan virtuaali-
                                // tuhoajafunktion määritys
```

Tarvitset tämän määrityksen, koska virtuaalituhoajafunktiot toimivat sillä tavalla, että viimeksi periytetyn luokan tuhoajafunktiota kutsutaan ensin ja sitten kutsutaan jokaisen kantaluokan tuhoajafunktioita. Tämä tarkoittaa sitä, että vaikka luokka on ab-

strakti, kääntäjät generoivat kutsun `~AWOV`-määritykseen. Sinun täytyy varmistaa, että lisäät funktion rungon. Jos et tee niin, linkkeri valittaa puuttuvasta symbolista ja sinun täytyy palata takaisin lisäämään se.

Siinä funktiossa voidaan tehdä mitä tahansa haluat, mutta kuten edellä mainitussa esimerkissä, on yleistä, että ei ole mitään tehtävissä. Jos näin on asia, sinua varmaan houkuttelee välttää tyhjään funktioon tapahtuvan kutsun kustannusten maksaminen esittelemällä tuhoajafunktio avoimena funktiona. Tämä on täydellisen järkevä strategia, mutta on olemassa ryppy, joka sinun pitää tietää.

Koska tuhoajafunktiosi on virtuaalinen, sen osoite täytyy kirjoittaa luokan `vtbl`-taulukkoon. Mutta avointen funktioiden ei pitäisi olla olemassa itsenäisinä funktioina (sitähän `inline`-avainsana tarkoittaa, eikö niin?), joten on ryhdyttävä erikoistoimiin osoitteen saamiseksi niille. Kohdassa 33 kerrotaan koko tarina, mutta ydin on tämä: jos julistat virtuaalisen tuhoajafunktion `inline`-tyyppisenä, tulet luultavasti välttämään funktion kutsun kustannukset silloin, kun sitä pyydetään avuksi, mutta kääntäjäsi täytyy silti myös luoda jonnekin ulkopuolella oleva funktio.

Kohta 15: `operator=`-funktion täytyy palauttaa viittaus `*this`-osoittimeen.

C++-kielen suunnittelija Bjarne Stroustrup joutui moniin vaikeuksiin varmistaessaan, että käyttäjän määrittelemät tyypit matkisivat sisäänrakennettuja tyyppejä mahdollisimman tarkoin. Voit tämän takia kuormittaa operaattoreita, kirjoittaa tyyppimuutoksen funktioita, ottaa hallintaan sijoittamisen ja kopiomuodostamisen ja niin edelleen. Koska hän joutui ponnistelemaan niin paljon, vähin mitä voit tehdä, on pitää pyörät pyörimässä.

Tästä pääsemmekin sijoittamiseen. Sisäänrakennettujen tyyppien avulla voit ketjuttaa sijoitukset yhteen, tähän tyyliin:

```
int w, x, y, z;
w = x = y = z = 0;
```

Tästä on tuloksena se, että sinun pitäisi pystyä ketjuttamaan yhteen myös käyttäjän määrittelemät tyypit:

```
string w, x, y, z;              // merkkijono on "käyttäjän
                                // C++-kielen normaalin
                                // kirjaston määrittelemä
                                // (katso Kohta 49)
w = x = y = z = "Hello";
```

Kohtalona on, että sijoitusoperaattori on oikealle assosiatiivinen, joten sijoitusketju jäsennetään näin:

```
w = (x = (y = (z = "Hello")));
```

Tämä kannattaa kirjoittaa täysin vastaavassa funktionaalisessa muodossa. Paitsi jos olet kaapissa oleva LISP-ohjelmoija, tulet varmaan kiitolliseksi tästä esimerkistä, koska sen avulla voit määrittää upotettuja operaattoreita:

```
w.operator=(x.operator=(y.operator=(z.operator=("Hello"))));
```

Tämä muoto on kuvaava, koska se korostaa sitä, että `w.operator=`, `x.operator=` ja `y.operator=` -funktioiden muuttuja on paluuarvo `operator=`-funktion edellisestä kutsusta. Tästä on tuloksena, että `operator=`-funktion palautustyypin täytyy olla hyväksyttävissä itse funktioon syöttötietona. Funktion allekirjoitus on C-luokan `operator=`-funktion oletusversiossa seuraava (katso Kohta 45):

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

Haluat varmasti noudattaa melkein aina seuraavaa sopimusta, jossa `operator=`-funktio sekä ottaa että palauttaa viittauksen luokkaolioon, vaikkakin voi olla, että funktio `operator=` kuormitetaan toisinaan niin, että se ottaa eri muuttujatyyppejä. Perus-`string`-tyyppi sisältää esimerkiksi kaksi erilaista versiota sijoitusoperaattorista:

```
string&                         // sijoita merkkijono
operator=(const string& rhs);   // merkkijonoon

string&                         // sijoita char*-tyyppinen
operator=(const char *rhs);     // merkkijonoon
```

Huomaa kuitenkin, että myös kuormitettaessa paluutyyppi on viittaus luokan olioon.

Uusien C++-ohjelmoijien yleinen virhe on ajatella, että `operator=`-funktio palauttaa `void`-operaattorin. Kyseessä on järkevältä tuntuva päätös, kunnes tajuat, että se estää sijoitusketjut. Joten älä tee niin.

Toinen yleinen virhe on se, että `operator=`-funktio palauttaa viittauksen `const`-tyyppiseen olioon, tähän tyyliin:

```
class Widget {
public:
  ...                                                  // huomaa
  const Widget& operator=(const Widget& rhs);          // const-
  ...                                                  // paluu-
};                                                     // tyyppi
```

Tavallisesti motivaationa on estää asiakkaita tekemästä tämän tyylisiä typeriä asioita:

```
Widget w1, w2, w3;
...
(w1 = w2) = w3;          // sijoita w2 -> w1, sitten w3 ->
                         // tulokseen! (Antamalla Widget-
                         // luokan operator=-funktiolle
                         // const-tyyppisen palautus-
                         // arvon estät tätä kääntymästä.)
```

Niin typerää kuin tämä onkin, se ei ole kiellettyä sisäänrakennetuille tyypeille:

```
int i1, i2, i3;
...
(i1 = i2) = i3;                // laillinen! sijoita i2 ->
                               // i1, sitten i3 -> i1!
```

En tiedä mitään käytännön hyötyä tämän kaltaisesta jutusta, mutta jos se kelpaa `int`-tyypeille, se kelpaa myös minulle ja luokilleni. Tämän pitäisi myös kelvata sinulle ja sinunlaisillesi. Miksi esitellä vastikkeettomia yhteensopimattomuuksia merkintätavoilla, joita sisäänrakennetut tyypit noudattavat?

Kun sijoitusoperaattori kantaa oletuksena olevan etumerkinnän, palautuvalle oliolle on kaksi vakavasti otettavaa ehdokasta: sijoituksen vasemmalla puolella oleva olio (se johon osoitetaan `this`-osoittimella) ja oikean käden puolella oleva olio (se, joka on nimetty parametrilistassa). Kumpi on oikein?

Tässä ovat `String`-luokan mahdollisuudet (luokka, jolle varmasti haluat kirjoittaa sijoitusoperaattorin, kuten selitin Kohdassa 11):

```
String& String::operator=(const String& rhs)
{
  ...
  return *this;                // palauta viittaus
                               // vasemmalla puolella
                               // olevaan olioon
}
String& String::operator=(const String& rhs)
{
  ...
  return rhs;                  // palauta viittaus
                               // oikealla puolella
                               // olevaan olioon
}
```

Tämä voi sinusta tuntua samalta kuin kuusi yksilöä vastaan puoli tusinaa, mutta tärkeitä eroavaisuuksia on.

Ensinnäkin `rhs`-viittauksen palauttava versio ei käänny. Tämä johtuu siitä, että `rhs` on viittaus `const`-tyyppiseen `String`-muuttujaan, mutta `operator=`-funktio palauttaa viittauksen `String`-muuttujaan. Kääntäjät eivät lopeta murheitasi, kun ne yrittävät palauttaa viittauksen ei-`const`-muuttujaan silloin, kun itse olio on `const`-tyyppinen. Ongelma voidaan kuitenkin ratkaista helposti - esittele `operator=`-funktio uudelleen tähän tyyliin:

```
String& String::operator=(String& rhs) { ... }
```

Asiakkaan lähdekoodi ei valitettavasti nyt käänny! Katso uudelleen alkuperäisen sijoitusketjun viimeiseen osaan:

```
x = "Hello";                    // same as x.op=("Hello");
```

Koska oikealla puolella oleva sijoitusmuuttuja ei ole oikeaa tyyppiä - se on `char`-taulukko, ei `String` - kääntäjien olisi pakko luoda tilapäinen `String`-olio (`String`-muodostinfunktion avulla) niin, että kutsu onnistuisi. Tämä tarkoittaa sitä, että heidän täytyisi muodostaa lähdetekstiä, joka olisi karkeasti vastaavaa:

```
const String temp("Hello");     // luo tilapäinen
x = temp;                       // välitä tilap. op=-funk.
```

Kääntäjät ovat halukkaita luomaan tällaisen tilapäisen muuttujan (paitsi jos tarvittava muodostinfunktio on tyypiltään `explicit` - katso Kohta 19), mutta huomaa, että tilapäinen olio on tyypiltään `const`. Tämä on tärkeää, koska se estää sinua siirtämästä tilapäistä muuttujaa vahingossa funktioon, joka muuttaa parametriään. Jos tämä olisi sallittua, ohjelmoijat olisivat yllättyneitä huomatessaan, että vain kääntäjän muodostama tilapäinen muuttuja oli muutettu, ei se argumentti, jonka he alunperin välittivät kutsupaikasta. (Tiedämme tämän, koska C++-kielen varhaiset versiot sallivat tämänkaltaisten tilapäisten muuttujien muodostamisen, välittämisen ja muuttamisen, ja tuloksena oli koko joukko yllättyneitä ohjelmoijia.)

Voimme nyt nähdä, miksi edellä mainittu asiakaskoodi ei käänny, jos `String`-muuttujan `operator=`-funktio on esitelty vastaanottamaan viittauksen `String`-muuttujaan, joka ei ole `const`-tyyppiä: ei ole koskaan laitonta välittää `const`-tyyppistä oliota funktiolle, joka epäonnistuu esitellessään vastaavan parametrin `const`-tyyppisenä. Tämä on vain yksinkertaista `const`-oikeellisuutta.

Huomaat olevasi siinä onnellisessa olotilassa, että sinulla ei ole mitään valinnanvaraa: haluat aina määrittää sijoitusoperaattorisi sellaisella tavalla, että ne palauttavat viittauksen niiden vasemmalla puolella olevaan argumenttiin `*this`. Jos teet jotain muuta, estät joko sijoitusketjut, kutsupaikoilta tapahtuvat implisiittiset tyyppimuunnokset tai molemmat.

Kohta 16: Sijoita `operator=`-funktion kaikkiin tietojäseniin.

Kohdassa 45 selvitetään, että C++ kirjoittaa sijoitusoperaattorin sinulle, jos et esittele sitä itse. Kohdassa 11 kuvataan, miksi et usein paljon välitä siitä, mitä se on sinulle kirjoittanut. Varmaan siis ihmettelet, voitko jotenkin hyötyä kolikon molemmista puolista, jolloin annat C++-kielen muodostaa oletuksena olevan sijoitusoperaattorin ja ylität valikoivasti ne kohdat, joista et pidä. Ei onnistu. Jos haluat ottaa hallintaasi minkä tahansa osan sijoitusprosessista, sinun täytyy tehdä koko asia itse.

Käytännössä tämä tarkoittaa sitä, että kun kirjoitat sijoitusoperaattoreita, sinun täytyy sijoittaa oliosi *jokaiseen* tietojäseneen:

```
template<class T>      // malli, jolla luokat kytketään
class NamedPtr {       // nimet osoittimiin (Kohta 12)
public:
  NamedPtr(const string& initName, T *initPtr);
  NamedPtr& operator=(const NamedPtr& rhs);

private:
  string name;
  T *ptr;
};

template<class T>
NamedPtr<T>& NamedPtr<T>::operator=(const NamedPtr<T>& rhs)
{
  if (this == &rhs)
    return *this;              // katso Kohta 17

  // assign to all data members
  name = rhs.name;             // sijoita nimeen

  *ptr = *rhs.ptr;             // sijoita osoittimeen se
                               // mihin osoitetaan,
                               // ei itse osoitinta

  return *this;                // katso Kohta 15
}
```

Tämä oli kyllä helppoa muistaa silloin, kun luokka alun perin kirjoitettiin, mutta yhtä tärkeää on, että sijoitusoperaattori(t) päivitetään, jos luokkaan lisätään uusia tietojäseniä. Jos esimerkiksi päätät päivittää `NamedPtr`-mallin niin, että se pitää mukanaan aikamerkinnän siitä, kun nimi muutetaan, sinun täytyy lisätä uusi tietojäsen, ja tämä vaatii sekä muodostinfunktion/-funktioiden että sijoitusoperaattorin/-operaattoreiden päivittämisen. Luokan päivittämisen ja uusien jäsenfunktioiden lisäämisen hulinassa ja huiskeessa on helppo antaa tämänkaltaisen asian unohtua.

Todellinen hauskanpito alkaa, kun periytyvyys tulee mukaan juhliin, koska periytetyn luokan sijoitusoperaattoreiden täytyy huolehtia myös kantaluokan jäsenten sijoittamisesta! Tutki tätä:

```
class Base {
public:
  Base(int initialValue = 0): x(initialValue) {}

private:
  int x;
};

class Derived: public Base {
public:
  Derived(int initialValue)
  : Base(initialValue), y(initialValue) {}

  Derived& operator=(const Derived& rhs);

private:
  int y;
};
```

`Derived`-olion sijoitusoperaattori voidaan kirjoittaa loogisesti tällä tavalla:

```
// virheellinen sijoitusoperaattori
Derived& Derived::operator=(const Derived& rhs)
{
  if (this == &rhs) return *this;    // katso Kohta 17

  y = rhs.y;                         // sijoita Derivedin
                                     // yks. tietojäseneen

  return *this;                      // katso Kohta 15
}
```

Valitettavasti tämä on väärin, koska sijoitusoperaattori ei vaikuta `Derived`-olion `Base`-osan `x`-tietojäseneen. Tutki esimerkiksi tätä lähdekoodin palasta:

```
void assignmentTester()
{
  Derived d1(0);               // d1.x = 0, d1.y = 0
  Derived d2(1);               // d2.x = 1, d2.y = 1

  d1 = d2;                     // d1.x = 0, d1.y = 1!
}
```

Huomaa, että sijoitus ei muuta `d1`-muuttujan `Base`-osaa.

Tämä ongelma voidaan korjata suoraviivaisesti tekemällä sijoitus `Derived::operator=`-funktion `x`-muuttujaan. Valitettavasti tämä on laitonta, koska `x` on `Base`-luokan yksityinen osa. Sinun täytyy sen sijaan tehdä eksplisiittinen sijoitus `Derived`-olion sijoitusoperaattorin sisältä sen `Base`-*osaan*.

Teet sen tällä tavalla:

```
// oikea sijoitusoperaattori
Derived& Derived::operator=(const Derived& rhs)
{
  if (this == &rhs) return *this;

  Base::operator=(rhs);       // this->Base::operator=
  y = rhs.y;

  return *this;
}
```

Teet tässä pelkästään eksplisiittisen kutsun `Base::operator=`-funktioon. Tämä kutsu, kuten kaikki jäsenfunktioista muihin jäsenfunktioihin kohdistuvat kutsut, käyttävät `*this`-osoitinta implisiittisenä vasemmanpuoleisena oliona. Tuloksena on, että `Base::operator=`-funktio tekee samat asiat kuin osoittimen `*this` `Base`-osassa - täsmälleen se vaikutus, jonka haluat.

Eräät kääntäjät valitettavasti hyljeksivät (virheellisesti) tämänkaltaista kutsua kantaluokan sijoitusoperaattoriin, jos kääntäjä on luonut tämän sijoitusoperaattorin (katso Kohta 45). Jos haluat lepytellä luopiomaisia kääntäjiä, sinun täytyy toteuttaa `Derived::operator=`-funktio tällä tavalla:

```
Derived& Derived::operator=(const Derived& rhs)
{
  if (this == &rhs) return *this;

  static_cast<Base&>(*this) = rhs;    // kutsu operator=
                                      // -funktiota
                                      // *this-osoittimen
                                      // Base-osassa
  y = rhs.y;

  return *this;
}
```

Tämä hirviömäisyys muuntaa `*this`-osoittimen viittaukseksi `Base`-luokkaan, ja suorittaa sitten sijoituksen muunnoksen lopputulokseen. Tämä tekee sijoituksen pelkästään `Derived`-olion `Base`-osaan. Kannattaa olla huolellinen! On tärkeää, että tuleva muunnos tapahtuu `Base`-olion *viittaukseen*, ei itse `Base`-olioon. Jos muunnat `*this`-osoittimen `Base`-olioksi, päädyt lopulta kutsumaan kopiomuodostinta `Base`-olioon, ja muodostamasi uusi olio tulee olemaan sijoituksen kohde; `*this` säilyy muuttumattomana. Tuskin haluat tätä.

Välittämättä siitä, minkä näistä työtavoista otat käyttöön, kun olet sijoittanut `Derived`-olion `Base`-osan, jatkat sitten `Derived`-olion sijoitusoperaattorilla tehden sijoitukset `Derived`-olion kaikkiin tietojäseniin.

Samankaltainen periytyvyydelle sukua oleva ongelma saa usein alkunsa siitä, kun toteutat periytetyn luokan kopiomuodostimia. Katso seuraavasta, mikä on kopiomuodostimen vastine koodille, jota juuri tutkimme:

```
class Base {
public:
  Base(int initialValue = 0): x(initialValue) {}
  Base(const Base& rhs): x(rhs.x) {}

private:
  int x;
};

class Derived: public Base {
public:
  Derived(int initialValue)
  : Base(initialValue), y(initialValue) {}

  Derived(const Derived& rhs)  // virheellinen kopio-
  : y(rhs.y) {}                // muodostin

private:
  int y;
};
```

`Derived`-luokka esittää yhden ilkeimmistä virheistä C++-kielen valtakunnassa: se epäonnistuu kantaluokkaosan kopioinnissa silloin, kun `Derived`-olio on kopiomuodostettu. Tällaisen `Derived`-olion `Base`-osa on tietenkin muodostettu, mutta se *on* muodostettu käyttämällä `Base`-olion *oletuksena* olevaa muodostinfunktiota. Sen jäsen x alustetaan arvolla 0 (oletuksena olevan muodostinfunktion oletuksena oleva parametriarvo), riippumatta kopioinnin kohteena olevan olion `x`:n arvosta!

Jos tämä ongelma halutaan välttää, `Derived`-olion kopiomuodostimen täytyy varmistaa, että `Base`-olion kopiointimuodostinta pyydetään avuksi `Base`-olion oletusmuodostinfunktion sijasta. Helppo homma. Varmista vain, että määrität alustusarvon `Base`-oliolle `Derived`-olion kopiomuodostimen jäsenen alustuslistassa:

```
class Derived: public Base {
public:
  Derived(const Derived& rhs): Base(rhs), y(rhs.y) {}

  ...

};
```

Nyt kun asiakas luo `Derived`-olion kopioimalla tätä tyyppiä olevan olemassaolevan olion, sen `Base`-osa kopioidaan myös.

Kohta 17: Tarkista itseensä kohdistuva sijoitus op=-funktiossa.

Itseensä kohdistuva sijoitus tapahtuu silloin, kun teet jotain tämänkaltaista:

```
class X { ... };
X a;
a = a;                          // a on sijoitettu itseensä
```

Tämä näyttää typerältä, mutta on täysin laillista, joten älä epäile hetkeäkään, etteivätkö myös ohjelmoijat tekisi niin. Vielä tärkeämpää on, että sijoitus itseensä voi ilmaantua tässä myönteiseltä näyttävässä muodossa:

```
a = b;
```

Jos `b` on toinen nimi `a`:lle (esimerkiksi viittaus, joka on alustettu `a`:han), niin silloin tämä on myös viittaus itseensä, vaikka se ei ulkonaisesti siltä näytäkään. Tämä on esimerkki *peitenimen käytöstä*: samalla perustana olevalla oliolla voi olla kaksi tai useampia nimiä. Kuten näet tämän Kohdan lopussa, peitenimen käyttö voi tulla esiin missä tahansa valepuvussa, joten sinun pitää ottaa se huomioon joka kerta, kun kirjoitat funktion.

On olemassa kaksi hyvää syytä, joiden takia kannattaa huolehtia siitä, että mahdollisesta peitenimien käytöstä sijoitusoperaattoreissa suoriudutaan. Vähäisempi niistä on tehokkuus. Jos tunnistat itseensä tapahtuvan sijoituksen sijoitusoperaattorisi/-operaattoreidesi yläosassa, voit palata takaisin välittömästi, säästäen mahdollisesti paljon työtä, jonka muuten joutuisit tekemään toteuttaessasi sijoituksen. Kohta 16 tähdentää, että esimerkiksi periytetyn luokan asianmukaisen sijoitusoperaattorin täytyy kutsua sijoitusoperaattoria jokaisen kantaluokan kohdalla, ja nämä luokat voivat itsekin olla periytettyjä luokkia, joten sijoitusoperaattorin rungon ohittaminen periytetyssä luokassa voi säästää suurelta joukolta muita funktiokutsuja.

Vielä tärkeämpi syy itseensä kohdistuvan sijoituksen tarkistamiseen on oikeellisuuden varmistaminen. Muista, että sijoitusoperaattorin täytyy tyypillisesti vapauttaa olion varaamat resurssit (toisin sanoen päästä eroon vanhasta arvosta), ennen kuin se voi varata uusia resursseja, jotka vastaavat sen uuteen arvoon. Kun sijoitus tapahtuu itseensä, tämä resurssien vapauttaminen voi olla tuhoisaa, koska vanhoja resursseja voidaan tarvita uusien varaamisen prosessissa.

Tutki `String`-tyyppisten olioiden sijoittamista, jossa sijoitusoperaattori epäonnistuu tarkistaessaan itsensä tapahtuvan sijoittamisen:

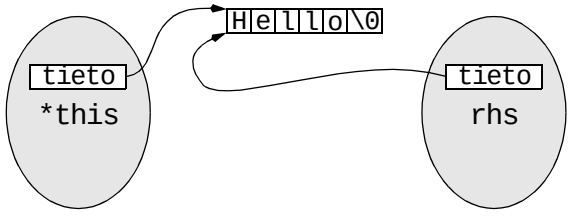
```
class String {
public:
  String(const char *value);       // katso Kohdasta 11
                                   // funktion määritys
  ~String();                       // katso Kohdasta 11
                                   // funktion määritys
  ...
  String& operator=(const String& rhs);
```

```
private:
  char *data;
};

// sijoitusoperaattori, joka jättää suorittamatta
// itseensä kohdistuvan sijoituksen tarkistuksen
String& String::operator=(const String& rhs)
{
  delete [] data;              // poista vanha muisti

  // varaa uutta muistia ja kopioi rhs:n arvo siihen
  data = new char[strlen(rhs.data) + 1];
  strcpy(data, rhs.data);

  return *this;                // katso Kohta 15
}
```

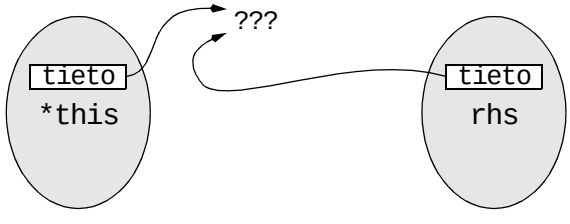
Tutki mitä tapahtuu nyt tässä tapauksessa:

```
String a = "Hello";
```

`*this` ja `rhs` tuntuvat olevan sijoitusoperaattorin sisällä olevia eri olioita, mutta tässä tapauksessa ne sattuvat olemaan eri nimiä samalle oliolle. Voit kuvata asian tällä tavalla:



Sijoitusoperaattori käyttää ensimmäiseksi `delete`-operaattoria `data`-osoittimeen ja tuloksena on seuraava tilanne:



Nyt kun sijoitusoperaattori yrittää suorittaa `strlen`-funktion `rhs.data`-osoittimeen, tulokset ovat määrittelemättömät. Tämä johtuu siitä, että `rhs.data` poistettiin samaan aikaan kuin `data`, mikä tapahtui siitä syystä, että `data, this->data` ja `rhs.data` ovat kaikki samaa osoitinta! Tästä lähtien asiat voivat vain mennä huonompaan suuntaan.

Nyt tiedät, että ratkaisu tähän vaikeaan pulmaan on tarkistaa itseensä kohdistuva sijoitus ja antaa paluuarvo välittömästi, jos tällainen sijoitus huomataan. Valitettavasti tämänkaltaisesta tarkistuksesta on helpompaa puhua kuin kirjoittaa, koska sinut pakotetaan heti hahmottamaan, mitä tarkoittaa se, että kaksi oliota ovat "samat".

Tämä aihe tunnetaan teknisesti olion identiteettinä (*object identity*), ja se on hyvin tunnettu aihe olio-piireissä. Tämä kirja ei ole esitelmä olion identiteetistä, mutta kaksi perustyötapaa kannattaa mainita ongelmaan.

Yksi työtapa on määrittää, että kaksi oliota ovat samat (niillä on sama identiteetti), jos niillä on sama arvo. Kaksi `String`-oliota voisivat esimerkiksi olla samat, jos ne esittäisivät saman merkkijonojakson:

```
String a = "Hello";
String b = "World";
String c = "Hello";
```

Tässä muuttujilla `a` ja `c` on sama arvo, joten niitä pidetään identtisinä; `b` eroaa näistä molemmista. Jos haluaisit käyttää tällaista identiteetin määritystä `String`-luokassasi, sijoitusoperaattorisi voisi näyttää tämänkaltaiselta:

```
String& String::operator=(const String& rhs)
{
  if (strcmp(data, rhs.data) == 0) return *this;

  ...

}
```

`operator==`-funktio määrittää yleensä samanarvoisuuden, joten yleinen muoto luokan `C`-luokan sijoitusoperaattorille, joka käyttää arvon samanarvoisuutta olion identiteetille, on tämä:

```
C& C::operator=(const C& rhs)
{
  // check for assignment to self
  if (*this == rhs)             // olettaa, että op== on
                                // olemassa
    return *this;

  ...

}
```

Huomaa, että tämä funktio vertaa *olioita* (`operator==`-funktion avulla), ei osoittimia. Kun käytetään arvojen samanarvoisuutta identiteetin määrittämisessä, ei ole mitään väliä sillä, valtaako kaksi oliota saman muistin; merkitystä on ainoastaan niiden esittämillä arvoilla.

Toinen mahdollisuus on pitää olion identiteettiä samanarvoisena sen muistissa olevan osoitteen avulla. Silloin, kun olioiden samanarvoisuudesta käytetään tätä määritystä, kaksi oliota ovat samat, jos ja vain jos niillä on sama osoite. Tämä määritys on yleisempi C++-ohjelmissa ehkä siksi, koska se on helppo toteuttaa ja laskeminen on nopeaa: näistä kumpikaan ei aina ole totta silloin, kun olion identiteetti perustuu arvoille. Silloin kun käytetään osoitteiden samanarvoisuutta, yleinen sijoitusoperaattori näyttää tältä:

```
C& C::operator=(const C& rhs)
{
  // tarkistetaan sijoitus itseensä
  if (this == &rhs) return *this;

  ...
}
```

Tämä riittää monille ohjelmille.

Jos tarvitset hienostuneempaa mekanismia määrittelemään, onko kaksi oliota sama, sinun täytyy toteuttaa se itse. Yleisin työtapa perustuu jäsenfunktioon, joka palauttaa eräänlaisen oliotunnistimen:

```
class C {
public:
  ObjectID identity() const;            // kts myös Kht 36

  ...
};
```

Oliot, joihin olio-osoittimet `a` ja `b` osoittavat, ovat identtiset, jos ja vain jos tapahtuu `a->identity() == b->identity()`. Olet tietenkin vastuussa siitä, että kirjoitat `operator==`-funktiot `ObjectID`-muuttujille.

Peitenimien käytön ja olion identiteetin ongelmat rajoittuvat tuskin `operator==`-funktioon. Se on vain se funktio, jossa tulet erityisen varmasti törmäämään niihin. Viittausten ja osoittimien läsnäollessa mitkä tahansa kaksi olion nimeä, jotka ovat yhteensopivaa tyyppiä, voivat viitata itse asiassa samaan olioon. Tässä on joitakin muita tilanteita, joissa peitenimien käyttö voi osoittaa Meduusa-tyyppistä ilmettä:

```
class Base {
  void mf1(Base& rb);           // rb ja *this voivat
                                // olla samat
  ...
};
void f1(Base& rb1,Base& rb2);   // rb1 ja rb2 voivat
                                // olla samat
```

```
class Derived: public Base {
  void mf2(Base& rb);          // rb ja *this voivat
                               // olla samat
  ...
};
int f2(Derived& rd, Base& rb); // rd ja rb voivat
                               // olla samat
```

Nämä esimerkit sattuvat käyttämään viittauksia, mutta osoittimet kelpaisivat myös aivan hyvin.

Kuten näet, peitenimien käyttö voi pompahtaa esiin monissa eri valepuvuissa, joten sitä ei pidä unohtaa, ja voit vain toivoa, että et koskaan törmää siihen. No, ehkä *voit*, mutta useimmat meistä eivät törmää. Senkin uhalla, että sotken kielikuvani, tämä on selvä osoitus siitä, että unssi ennaltaehkäisyä on kullan arvoista. Joka kerta, kun kirjoitat funktion, jossa peitenimien käyttö voi ajateltavissa olevasti olla läsnä, sinun *täytyy* ottaa huomioon tämä mahdollisuus.