14

Luokan alustus, sijoitus ja tuhoaminen

Tässä luvussa katsotaan yksityiskohtaisesti luokkaolioiden automaattista alustamista, sijoitusta ja tuhoamista ohjelman aikana. Alustamista tukee *muodostaja* (*constructor,* myös *muodostin*, *konstruktori*). Muodostaja on mahdollisesti ylikuormitettu, käyttäjän määrittelemä funktio, jonka luokan suunnittelija on luonut ja jota käytetään automaattisesti jokaiseen luokkaolioon ennen niiden ensimmäistä käyttöä ohjelmassa. Täydentävää käyttäjän määrittelemää jäsenfunktiota, *tuhoajaa* (*destructor,* myös *destruktori*), käytetään automaattisesti luokan jokaiseen luokkaolioon niiden viimeisen käytön jälkeen. Tuhoajaa käytetään pääasiassa niiden resurssien vapauttamiseen, jotka on hankittu joko luokan muodostajalla tai luokan elinkaaren aikana.

Oletusarvo on, että tuetaan yhden luokan sekä sellaista alustamista että sijoittamista toiseen (*default memberwise semantics*), jossa luokan jokainen tietojäsen kopioidaan vuorollaan. Vaikka tämä riittää yleistapauksissa, on oletusarvo riittämätön tietyissä tilanteissa luokan turvallisuuden ja käsittelyn oikeellisuuden kannalta. Näissä tapauksissa luokan suunnittelijan pitää tehdä *kopiointimuodostajan* ja *kopioinnin sijoitusoperaattorin* määrittelyt. Usein on kaikkein vaikein asia havaita, että nämä erityiset jäsenfunktiot on tarpeen tehdä.

14.1 Luokan alustus

Mietitäänpä seuraavaa luokan määrittelyä:

```
class Data {
public:
    int ival;
    char *ptr;
};
```

Jotta tämän luokan oliota voitaisiin käyttää turvallisesti, pitää varmistua, että sen kaksi jäsentä alustetaan oikein. Mitä se milloinkin tarkoittaa, vaihtelee kuitenkin luokasta toiseen. Pitäisikö esimerkiksi sallia, että ival voisi sisältää negatiivisen tai nolla-arvon? Mitkä ovat sen kahden jäsenen oikeat alustusarvot? Emme voi sanoa sitä, ennen kuin ymmärrämme abstraktion, jota luokan on tarkoitus edustaa. Jos sen on tarkoitettu edustavan jonkin yrityksen työntekijää, silloin ptr asetetaan todennäköisesti viittaamaan työntekijän nimeen ja ival yksilölliseen työntekijän numeroon. Negatiivinen numero tai nolla olisi kelpaamaton. Vaihtoehtoisesti, jos luokka edustaa vaikkapa tämänhetkistä kaupungin ilmalämpötilaa, silloin joko negatiivinen tai positiivinen numero tai nolla olisi kelvollinen. Toinen mahdollisuus on, että Data edustaa *merkkijonoa, johon kohdistuvat viittaukset on laskettu*: ival-arvo on tämänhetkinen viittausten lukumäärä merkkijonoon, johon ptr osoittaa. Tässä abstraktiossa ival alustetaan alkuarvolla 1. Jos tuo arvo putoaa nollaan, luokkaolion varaama muisti on vapautettu.

Luokan ja sen kahden tietojäsenen mnemoniset nimet saisivat tietysti luokan aikomuksen selvemmäksi ohjelman lukijoille, mutta se vaatisi lisätietoja ohjelman kääntäjälle. Jotta kääntäjä ymmärtäisi aikomuksemme, pitää tehdä yksi tai joukko erityisiä ylikuormitettuja *muodostaja*-alustusfunktioita. Sopiva muodostaja valitaan olion määrittelyssä ilmaistujen alkuarvojen perusteella. Esimerkiksi seuraavista jokainen voi edustaa kelvollista ja yksilöllistä Data-luokkaolion alustusta:.

```
Data dat01( "Venus and the Graces", 107925 );
Data dat02( "about" );
Data dat03( 107925 );
Data dat04;
```

Ohjelmissamme on tilanteita (kuten dat04), joissa tarvitsemme luokkaolion, mutta emme vielä tiedä, mikä alkuarvon pitäisi olla. Ehkä nämä arvot voi päätellä vasta myöhemmin. Silti meidän pitää antaa joku alkuarvo vain ilmaistaksemme, että mitään arvoa ei ole vielä valittu. Eräässä mielessä on joskus tarpeen alustaa luokan olio ilmaistaksemme, että sitä *ei vielä* ole alustettu. Useimmissa luokissa on *oletusmuodostaja*, joka ei vaadi alkuarvojen määritystä. Tyypillinen oletusmuodostaja alustaa luokan olion niin, että voimme jatkossa havaita sen olevan alustamatta.

Pitääkö Data-luokallemme tehdä muodostaja? Sattumalta ei, koska sen kaikki tietojäsenet ovat julkisia. C-kielestä peritty mekanismi tukee eksplisiittistä alustusluetteloa, joka muistuttaa taulukon alustusta. Esimerkiksi:

```
int main()
{
        // local1.ival = 0; local1.ptr = 0
        Data local1 = { 0, 0 };

        // local2.ival = 1024;
        // local2.ptr = "Anna Livia Plurabelle"
```

```
    Data local2 = { 1024, "Anna Livia Plurabelle" };

    // ...
}
```

Arvot ratkaistaan paikkasidonnaisesti, mikä perustuu tietojäsenten esittelyjärjestykseen. Esimerkiksi seuraava saa aikaan käännösvirheen, koska ival on esitelty ennen ptr:ää:

```
// virhe: ival = "Anna Livia Plurabelle"
//       ptr  = 1024
Data local2 = { "Anna Livia Plurabelle" , 1024 };
```

Eksplisiittisen alustusluettelon kaksi haittapuolta on, että sitä voidaan käyttää luokan vain sellaisiin olioihin, joiden kaikki tietojäsenet ovat julkisia (tarkoittaa, että eksplisiittinen alustusluettelo ei tue tiedon kapselointia eikä abstrakteja tietotyyppejä — nämä puuttuvat C-kielestä, josta tämä alustuksen muoto on johdettu), ja että se vaatii ohjelmoijan välitöntä puuttumista asiaan ja siten lisää vahingon (alustusluettelon unohtaminen) tai virheen (jotenkin sekoitetaan alustusjärjestys) mahdollisuutta.

Kun haittapuolet otetaan huomioon, voidaanko eksplisiittisen alustusluettelon käyttöä muodostajan asemesta puolustella koskaan? Käytännössä kyllä. Joissakin sovelluksissa on tehokkaampaa käyttää eksplisiittistä alustusluetteloa, kun alustetaan suuria tietorakenteita vakioarvoilla. Olemme ehkä rakentamassa esimerkiksi väripalettia tai ahdamme ohjelmatekstiin suuria määriä vakioarvoja kuten monimutkaisen geometrisen mallin ohjaus- ja solmuarvoja. Näissä tapauksissa voidaan eksplisiittinen alustus suorittaa ohjelmalatauksen aikana, jolloin säästetään muodostajan käynnistysaikaa, vaikka se olisi määritelty välittömäksi etenkin globaaleille olioille[1].

Yleensä on kuitenkin pidetty muodostajaa parempana luokan alustusmekanismina, josta taataan, että kääntäjä käyttää sitä luokan jokaiseen olioon ennen niiden ensimmäistä käyttöä. Seuraavassa kohdassa katsomme luokan muodostajaa yksityiskohtaisemmin.

14.2 Luokan muodostaja

Muodostaja tunnistetaan antamalla sille sama nimi kuin luokalla on. Kun esittelemme oletusmuodostajan, kirjoitamme[2]

```
class Account {
public:
    // oletusmuodostaja ...
  Account();
  // ...
private:
    char *_name;
```

1. Katso julkaisusta [LIPPMAN96a] tarkempi käsittely esimerkkeineen ja karkeine suorituskykyaikoineen.
2. Normaalisti esittelisimme _name:n string-tyyppiseksi. Esittelemme sen C-tyyliseksi merkkijonoksi, jotta voimme lykätä luokan tietojäsenten alustuskysymykset kohtaan 14.4.

```
    unsigned int _acct_nmbr;
    double _balance;
};
```

Muodostajan ainoa syntaktinen rajoitus on, että sille ei saa määrittää paluutyyppiä, ei edes void-tyyppiä. Esimerkiksi molemmat seuraavat esittelyt ovat virheellisiä:

```
// virheitä: muodostajalle ei saa määrittää paluuarvoa
void    Account::Account() { ... }
Account* Account::Account( const char *pc ) { ... }
```

Ei ole rajoitusta muodostajien lukumäärälle, joita voimme esitellä luokalle, edellyttäen, että jokaisen muodostajan parametriluettelo on yksilöllinen.

Kuinka tiedämme, millaisia tai kuinka monta muodostajaa määrittelemme? Vähintään meidän tulee sallia, että käyttäjä voi antaa alkuarvon jokaiselle tietojäsenelle, joka pitää asettaa. Esimerkiksi tilinumero voidaan asettaa tai generoida automaattisesti, jotta sen yksilöllisyys voidaan taata. Sanokaamme, että meidän tapauksessamme se on generoitu automaattisesti. Tämä jättää meille tarpeen, että mahdollistamme kahden jäsenen, _name ja _balance, alustamisen:

```
Account( const char *name, double open_balance );
```

Account-olio, joka alustetaan tällä muodostajalla, voidaan määritellä seuraavasti:

```
Account newAcct( "Mikey Metz", 0 );
```

Vaihtoehtoisesti, jos on paljon tilejä, jotka aloitetaan avaamalla nollasaldolla, voi käyttäjä pyytää määrittämään ainoastaan nimen ja antaa muodostajan alustaa _balance nollaksi automaattisesti. Eräs ratkaisu on tehdä toinen muodostaja, joka on muotoa

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

Vaihtoehtoinen ratkaisu on tehdä nolla-oletusarvo kaksiparametriseen muodostajaan:

```
Account( const char *name, double open_balance = 0.0 );
```

Molemmat muodostajat tarjoavat käyttäjän pyytämää toiminnallisuutta ja siinä mielessä kumpikin ratkaisu on hyväksyttävä. Mielestämme parempi ratkaisu on oletusargumentti, koska se eliminoi luokkaan liittyvien muodostajien lukumäärää.

Pitäisikö meidän tukea myös sitä, että avataan saldo, mutta ei anneta asiakkaan nimeä? Sattumalta luokan määritys ei salli tätä eksplisiittisesti. Kaksiparametrinen muodostajamme toisella oletusargumentillaan muodostaa täydellisen rajapinnan, jolle voidaan antaa Account-luokan tietojäsenille alkuarvoja, joita käyttäjä voi asettaa:

```
class Account {
public:
        // oletusmuodostaja ...
            Account();

            // parametrien nimet eivät ole välttämättömiä esittelyssä
            Account( const char*, double=0.0 );

            const char* name() { return _name; }
            // ...
private:
            // ...
};
```

Molemmat seuraavat ovat sallittuja Account-luokan olioiden määrittelyjä, joissa välitetään yksi tai kaksi argumenttia muodostajalle:

```
int main()
{
            // ok: molemmat käynnistävät kaksiparametrisen muodostajan
            Account acct( "Ethan Stern" );
            Account *pact = new Account( "Michael Lieberman", 5000 );

            if ( strcmp( acct.name(), pact->name() ))
                // ...
}
```

C++ vaatii, että muodostajaa pitää käyttää luokkaolioon ennen sen ensimmäistä käyttöä. Tämä tarkoittaa, että sekä `acct`-oliolle että oliolle, jota `pact` osoittaa, pitää käyttää muodostajaa ennen `if`-lauseen ehdon testausta.

Sisäisesti kääntäjä kirjoittaa ohjelmamme uudelleen ja lisää muodostajan käynnistyksen. Seuraavassa kuvataan, kuinka `main()`-funktiossa `acct`-olion määrittelyä todennäköisesti kasvatetaan:

```
// C++-pseudokoodia
// kuvaus muodostajan sisäisestä lisäyksestä

int main()
{
     Account acct;
     acct.Account::Account("Ethan Stern", 0.0);
  // ...
}
```

Tietysti, jos muodostajan ilmentymä on määritelty välittömäksi, se laajennetaan käynnistyskohtaan.

New-lausekkeen käsittely on hieman monimutkaisempi. Muodostaja käynnistetään vain, jos new-lauseke onnistuu saamaan muistia. pact-olion määrittelyn kasvatus voisi todennäköisesti näyttää tältä (tätä on hieman yksinkertaistettu):

```
// C++-pseudokoodia
// muodostajan lisäys, kun käytetään new-lauseketta
int main()
{
        // ...

            Account *pacct;
        try {
                        pact = _new( sizeof( Account ));
                        pact->Acct.Account::Account(
                                "Michael Lieberman", 5000.0);

            }
            catch( std::bad_alloc ) {
                        // new-operaattori epäonnistui
                        // muodostajaa ei koskaan suoriteta
            }
            // ...
}
```

On olemassa kolme yleisesti yhtäläistä muotoa argumenttien määrittämiseen muodostajalle:

```
// yleisesti yhtäläiset muodot
Account acct1( "Anna Press" );
Account acct2 = Account( "Anna Press" );
Account acct3 = "Anna Press";
```

Muotoa acct3 voidaan käyttää vain, kun määritetään yksi argumentti. Kahdelle tai useammalle argumentille voidaan käyttää vain muotoja acct1 ja acct2. Yleensä suosittelemme muodon acct1 käyttöä:

```
// suositeltu muodostajan muoto
Account acct1( "Anna Press" );
```

Yleinen virhe kielen uusille käyttäjille on esitellä olio alustamalla se oletusmuodostajalla seuraavasti:

```
// hups! tämä ei toimi kuten oli tarkoitus
Account newAccount();
```

Tämä kääntyy virheittä. Mutta, kun yritämme käyttää sitä kuten seuraavassa

```
// käännöksenaikainen virhe
if ( ! newAccount.name() ) ...
```

kääntäjä valittaa, että ei voi käyttää jäsenen käsittelyn merkintätapaa funktioon! Kääntäjä tulkitsee määrittelyn

```
// määrittelee newAccount-funktion
// eikä Account-luokan oliota
Account newAccount();
```

niin, että siinä määritellään funktio, joka ei saa parametreja ja palauttaa Account-tyyppisen olion — tuskin sitä, mitä aioimme! Luokkaolion oikea esittelytapa, joka käyttää oletusmuodostajaa, on jättää pois lopusta tyhjä sulkupari:

```
// ok: määrittelee luokan olion ...
Account newAccount;
```

Luokkaolio voidaan määritellä ilman argumentteja vain, jos mitään muodostajia ei ole esitelty tai on esitelty oletusmuodostaja. Kun luokka esittelee yhden tai useamman muodostajan, ei luokan oliota voi muodostaa ilman, ettei se käynnistä yhtä tai useampaa muodostajan ilmentymää. Etenkin, jos luokka esittelee muodostajan, joka saa yhden tai useamman parametrin, mutta ei esittele oletusmuodostajaa, pitää jokaiselle luokan olion määrittelylle antaa vaaditut argumentit. Projektissamme voi joku esimerkiksi väittää, että ei ole järkeä määritellä oletusmuodostajaa Account-luokalle, koska jokaisella kelvollisella tilillä pitää olla käyttäjän nimi. Uudistettu Account-luokka voisi sitten poistaa oletusmuodostajan:

```
class Account {
public:
        // parametrien nimet eivät ole välttämättömiä esittelyssä
        Account( const char*, double=0.0 );

        const char* name() { return _name; }
        // ...
private:
        // ...
};
```

Nyt jokaisen Account-luokan olion *pitää* antaa vähintään C-tyylinen merkkijonoargumentti luokan muodostajalle, jotta se olisi kelvollinen luokkaolion määrittely. Vaikka tämä saattaa kiinnittää meidät tiukemmin Account-luokan abstraktion määritykseen, tämä voi käytännössä osoittautua epäkäytännölliseksi. Miksi? Säiliöluokat (kuten esimerkiksi vektori) vaativat, että niiden elementeille annetaan joko oletusmuodostaja tai ei muodostajaa ollenkaan. Samalla tavalla luokkaolioiden dynaaminen taulukko vaatii joko oletusmuodostajaa tai ei muodostajaa ollenkaan. Esimerkiksi juuri määrittelemämme Account-luokkamme saa aikaan virheen, jos käyttäjämme kirjoittaisi seuraavan:

```
// virhe: vaatii Account-luokan oletusmuodostajan
Account *pact = new Account[ new_client_cnt ];
```

Käytännössä on melkein aina välttämätöntä tehdä oletusmuodostaja, jos määritellään myös muita muodostajia.

Mitä, jos ei ole olemassa sopivia oletusarvoja luokalle? Esimerkiksi Account-luokkamme vaatii, että oliolle määritetty nimi on kelvollinen. Tässä tapauksessa ainoa asia, jonka voimme tehdä, on alustaa olio ilmaistaksemme, että sitä ei ole vielä alustettu kelvollisilla arvoilla. Esimerkiksi:

```
// Account-luokan oletusmuodostaja
inline Account::
Account() {
    _name = 0;
    _balance = 0.0;
    _acct_nmbr = 0;
}
```

Sitten meidän pitää ohjelmoida tarkistuksia Account-luokan jäsenfunktioihin taataksemme olioiden yhtenäisyyden ennen niiden käyttöä.

Luokan jäsenen alustamiselle on olemassa vaihtoehtoinen syntaksi: *jäsenen alustusluettelo*, joka on pilkuin eroteltu luettelo jäsenten nimiä alkuarvoineen. Esimerkiksi Account-luokan oletusmuodostaja voidaan kirjoittaa uudelleen seuraavasti:

```
// Account-luokan oletusmuodostaja, joka käyttää
// jäsenen alustusluetteloa
inline Account::
Account()
        : _name( 0 ),
          _balance( 0.0 ), _acct_nmbr( 0 )
{}
```

Jäsenen alustusluettelo voidaan määrittää vain muodostajan määrittelyyn, ei sen esittelyyn. Alustusluettelo sijoitetaan parametriluettelon ja muodostajan rungon väliin. Se alkaa kaksoispisteellä. Tässä on kaksiparametrinen muodostajamme, joka käyttää osittain hyväkseen jäsenen alustusluetteloa:

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

    _acct_nmbr = get_unique_acct_nmbr();
}
```

`get_unique_acct_nmbr()` on muu kuin julkinen funktio, joka palauttaa tilinumeron, josta taataan, että se ei ole käytössä.

Muodostajaa ei pidä esitellä const- tai volatile-avainsanalla (katso kohta 13.3.5). Esimerkiksi ei ole sallittua kirjoittaa seuraavaa kummallakaan tavalla:

```
class Account{
public:
    Account() const;    // virhe
    Account() volatile; // virhe
    // ...
};
```

Ilmeisesti tämä ei voi merkitä, että luokan const- tai volatile-olioita ei voisi alustaa muodostajalla. Sen sijaan luokan oliolle voidaan käyttää sopivaa muodostajaa riippumatta siitä, onko sen tyyppi const, ei-const vai volatile. Luokkaolion const-tyyppisyys tulee voimaan, kun muodostaja lopettaa suorituksensa ja olio on alustettu. const-tyyppisyys häviää, kun tuhoaja on käynnistetty. Luokan const-oliota pidetään siten const-tyyppisenä aikana, jolloin sen muodostaja päättyy ja tuhoaja aloittaa. Sama pätee myös luokan volatile-olioon.

Mietitäänpä seuraavaa koodikatkelmaa:

```
// jossain otsikkotiedostossa
extern void print( const Account &acct );

// ...

int main()
{
    // konvertoi "hups" Account-olioksi
  // käyttäen Account::Account( "hups", 0.0 )
  print( "hups" );

    // ...
}
```

Oletusarvo on, että yksiparametrinen muodostaja (tai moniparametrinen muodostaja oletusarvoineen kaikille muille paitsi ensimmäiselle parametrille) toimii konversio-operaattorina. Koodikatkelmassa kääntäjä käyttää Account-muodostajaa implisiittisesti konvertoidakseen merkkijonoliteraalin Account-olioksi print()-funktion käynnistyksessä, vaikka konversio ei sovi tähän tilanteeseen.

Tahattomat implisiittiset luokkakonversiot kuten konversio merkkijonosta "hups" Account-olioksi on osoittautunut yleiseksi ohjelmointivirheeksi, jota on vaikea jäljittää. Avainsana explicit esiteltiin C++-standardiin, jotta se helpottaisi tämän epämieluisan kääntäjävirheen ehkäisyä. explicit-määre ilmaisee kääntäjälle, ettei se käyttäisi implisiittisiä konversioita:

```
class Account{
public:
        explicit Account( const char*, double=0.0 );
        // ...
};
```

explicit-avainsanaa voidaan käyttää vain muodostajaan. (Konversio-operaattorit ja explicit-avainsana on käsitelty kohdassa 15.9.2.)

14.2.1 Oletusmuodostaja

Oletusmuodostaja on sellainen, joka voidaan käynnistää ilman käyttäjän määrittämiä argumentteja. Tämä ei tarkoita, ettei se voi hyväksyä argumentteja. Se tarkoittaa vain, että muodostajassa on jokaiselle parametrille vastaava oletusarvo. Esimerkiksi jokainen seuraavista edustaa oletusmuodostajaa:

```
// jokainen on oletusmuodostaja
Account::Account() { ... }
iStack::iStack( int size = 0 ) { ... }
Complex::Complex(double re=0.0,double im=0.0) { ... }
```

Kun kirjoitamme

```
int main()
{
    Account acct;
    // ...
}
```

kääntäjä tarkistaa ensiksi, onko Account-luokalle määritelty oletusmuodostajaa. Jokin seuraavista tapahtuu:

1. Oletusmuodostaja on määritelty. Sitä käytetään acct-olioon.
2. Oletusmuodostaja on määritelty, mutta se ei ole julkinen. acct-olion määrittely saa aikaan käännöksenaikaisen virheen: main()-funktiolla ei ole pääsyoikeutta.
3. Oletusmuodostajaa ei ole määritelty, vaan yksi tai useampi muodostaja, jotka vaativat argumentteja. acct-olion määrittely saa aikaan käännöksenaikaisen virheen: liian vähän argumentteja muodostajalle.
4. Oletusmuodostajaa ei ole määritelty eikä muita muodostajia. Määrittely ei ole sallittu. acct on alustamatta eikä muodostajaa käynnistetä.

Kohdat 1 ja 3 tulisi kohtuudella ymmärtää tässä vaiheessa (jos ei, vilkaise taaksepäin tätä

lukua alusta alkaen tähän saakka — toivottavasti tämä ei johda lukijan loputtomaan lukusilmukkaan!). Katsokaamme hieman tarkemmin kohtia 2 ja 4.

Jos Account-luokkamme esittelee kaikki jäsenensä julkisiksi eikä esittele muodostajaa kuten seuraavassa

```
class Account{
public:
    char        *_name;
    unsigned int  _acct_nmbr;
    double        _balance;
};
```

silloin jokaisen Account-luokkaolion määrittely johtaa tilanteeseen, jossa luokkakohtaista alustusta ei tapahdu. Kolmen jäsenen alkuarvot riippuvat jokaisen olion määrittelyn yhteydestä. Olioille, joilla on staattinen ulottuvuus kuten seuraavassa

```
// staattinen ulottuvuus
// jokaiseen olioon liittyvä muisti
// nollataan

Account global_scope_acct;
static Account file_scope_acct;

Account foo()
{
    static Account local_static_acct;
    // ...
}
```

taataan, että niiden jäsenet “nollataan” (sama pätee luokattomille olioille).

Olioille, jotka on määritelty joko paikallisesti tai varattu dynaamisesti, taataan, että niiden muistitila täytetään alkuarvoisesti satunnaisilla biteillä, jotka muodostuvat ohjelman suorituksenaikaisen pinon aikaisemman käytön perusteella. Esimerkiksi:

```
// paikalliset ja keosta varatut oliot ovat alustamatta,
// kunnes ne alustetaan tai niihin sijoitetaan

Account bar()
{
    Account local_acct;
    Account *heap_acct = new Account;
    // ...
}
```

Vasta-alkajat erehtyvät usein uskomaan, että kääntäjä generoi ja käyttää oletusmuodostajaa automaattisesti, jos sellaista ei ole tehty — ja alustaa luokan tietojäsenet. Account-luokallemme tämä ei yksinkertaisesti päde, koska määrittelimme sen niin. Yhtään oletusmuodostajaa ei generoida tai käynnistetä. Monimutkaisemmille luokille, jotka sisältävät tietojäseniä tai käyttävät hyväkseen periytymistä, tämä on osittain totta: oletusmuodostaja saatetaan generoida, mutta se

ei anna alkuarvoja sisäisille tyypeille tai yhdistetyille tyypeille kuten osoittimet ja taulukot.

Jos haluamme alustaa luokan sisäisiä tietojäseniä tai yhdistettyjä tyyppejä, pitää se tehdä yhdessä tai useammassa muodostajassa. Ellemme tee niin, on melkein mahdotonta erottaa toisistaan kelvollinen ja alustamaton arvo, joka liittyy luokkaolion paikallisesti tai dynaamisesti varattuun sisäiseen tietojäseneen tai yhdistettyyn tyyppiin[3].

14.2.2 Olioiden muodostamisen rajoittaminen

Muodostajan käsiteltävyys päätellään käsittelyosasta, jossa se on esitelty. Voimme rajoittaa tai eksplisiittisesti kieltää tietynmuotoisten olioiden muodostamisen sijoittamalla vastaavan muodostajan muuhun kuin julkiseen käsittelyosaan. Tässä esimerkissä Account-oletusmuodostaja on esitelty yksityiseksi ja kaksiparametrinen muodostaja on esitelty julkiseksi:

```
class Account {
    friend class vector< Account >;
  public:
      explicit Account( const char*, double = 0.0 );
      // ...
  private:
      Account();
      // ...
};
```

Yleinen ohjelma voi määritellä vain Account-olioita joko nimellä tai nimellä ja avattavalla tilin saldolla. Account-luokan jäsenfunktiot ja sen ystävä, vektoriluokka, voivat määritellä Account-olioita käyttämällä kummankin muodostajan rajapintaa.

Yksityisten muodostajien vallitsevat käyttösyyt todellisissa C++-ohjelmissa ovat

1. estää omien luokkaolioiden kopiointi toisiinsa (käsitellään seuraavassa alikohdassa) ja
2. ilmaista, että muodostaja aiotaan käynnistää vain, kun luokka toimii kantaluokkana periytymishierarkiassa eikä oliona, jota voidaan käsitellä suoraan ohjelmasta (katso keskustelu periytymisestä ja oliosuuntautuneesta ohjelmoinnista luvusta 17).

3. Tämä niille, jotka ovat ohjelmoineet C:llä aikaisemmin: Account-määrittely käyttäytyy täsmälleen samoin kuin jos olisimme kirjoittaneet C:llä:

```
typedef struct {
    char        *_name;
    unsigned int  _acct_nmbr;
    double        _balance;
} Account;
```

14.2.3 Kopiointimuodostaja

Luokkaolion alustamista oman luokkansa toisella oliolla sanotaan *jäsenittäiseksi oletusalustukseksi* (*default memberwise initialization*). Teoriassa luokkaolion kopiointi toiseen toteutetaan kopioimalla jokainen ei-staattinen tietojäsen vuorollaan. Luokan suunnittelija voi korvata oletuskäyttäytymisen tekemällä erityisen luokan *kopiointimuodostajan* (*copy constructor*). Jos sellainen on määritelty, se käynnistetään aina, kun luokkaolio alustetaan oman luokkansa toisella oliolla.

Jäsenittäinen oletusalustus on usein sopimaton luokkamme oikealle käyttäytymiselle. Korvaamme oletuskäyttäytymisen määrittelemällä eksplisiittisen ilmentymän kopiointimuodostajasta. Account-luokkamme vaatii, että teemme sellaisen, koska muutoin Account-olioilla olisi samoja tilinumeroita, joita ei nimenomaan sallita luokan määrityksissä.

Kopiointimuodostaja saa muodollisen viittausparametrin luokkaolioon (esitellään perinteisesti const-tyyppinä). Tässä on toteutuksemme:

```
inline Account::
Account( const Accout &rhs )
        : _balance( rhs._balance )
{
    _name = new char[ strlen(rhs._name)+1 ];
    strcpy( _name, rhs._name );

    // ei saa kopioida tätä: rhs._acct_nmbr
    _acct_nmbr = get_unique_acct_nmbr();
}
```

Kun kirjoitamme

```
Account acct2( acct1 );
```

kääntäjä päättelee, onko Account-luokalle esitelty eksplisiittinen kopiointimuodostaja. Jos se on esitelty ja käytettävissä, se käynnistetään. Jo se on esitelty, mutta siihen ei päästä käsiksi, saa acct2-määrittely aikaan käännöksenaikaisen virheen. Jos kopiointimuodostajan ilmentymää ei ole esitelty, toteutetaan jäsenittäinen oletusalustus. Jos jälkeenpäin esittelemme tai poistamme kopiointimuodostajan esittelyn, ei käyttäjän ohjelmia tarvitse muuttaa. Ne pitää kuitenkin kääntää uudelleen. (Kohdassa 14.6 katsotaan jäsenittäistä alustamista tarkemmin.)

Harjoitus 14.1

Mitkä seuraavista lauseista eivät ole totta, vai onko yksikään? Miksi?

(a) Luokalla pitää olla ainakin yksi muodostaja.

(b) Oletusmuodostaja on sellainen, jolla ei ole parametreja parametriluettelossa.

(c) Ellei luokalle ole merkityksellisiä oletusarvoja, ei luokalle pitäisi tehdä oletusmuodostajaa.

(d) Ellei luokalla ole oletusmuodostajaa eksplisiittisesti, kääntäjä generoi sellaisen automaattisesti ja alustaa jokaisen tietojäsenen tyyppinsä mukaisella oletusarvolla.

Harjoitus 14.2

Tee yksi tai useampi muodostaja seuraaville tietojäsenille. Perustele valintasi.

```
class NoName{
public:
    // muodostaja(t) tulee tähän ...
  // ...
protected:
    char  *pstring;
    int    ival;
    double dval;
};
```

Harjoitus 14.3

Valitse yksi seuraavista abstraktioista (tai valitse oma). Päättele, millainen tieto (jonka käyttäjät voivat asettaa) on sopiva luokalle. Tee sopiva muodostajajoukko. Perustele päätöksesi.

(a) Kirja

(b) Päivämäärä

(c) Työntekijä

(d) Ajoneuvo

(e) Olio

(f) Puu

Harjoitus 14.4

Kun käytetään seuraavaa Account-luokan määrittelyä

```
class Account{
public:
    Account();
    explicit Account( const char*, double=0.0 );
    // ...
};
```

selitä, mitä tapahtuu seuraavissa määrittelyissä:

```
(a) Account acct;
(b) Account acct2 = acct;
(c) Account acct3 = "Rena Stern";
(d) Account acct4( "Anna Engel", 400.00 );
(e) Account acct5 = Account( acct3 );
```

Harjoitus 14.5

Kopiointimuodostajan parametrin ei ole pakko olla const-tyyppinen, mutta sen on pakko olla viittaus. Miksi seuraava on virheellinen?

```
Account::Account( const Account rhs );
```

14.3 Luokan tuhoaja

Muodostajan eräs tarkoitus on tehdä resurssin hankinta automaattiseksi. Näimme tästä esimerkin Account-luokkamme muodostajassa, jossa merkkitaulukko varataan new-lausekkeen avulla ja yksilöllinen tilinumero varmistetaan. Vaihtoehtoisesti voisimme haluta lukita alueen jaetusta muistista tai ohjelmasäikeen kriittisestä osasta. Se, mitä puuttuu, on symmetrinen operaatio, joka tekee automaattisen vapautuksen eli palauttaa luokkaolioon liittyvän resurssin sen elinkaaren päätyttyä. *Tuhoaja* (*destructor*) on juuri sellainen erityinen luokan jäsenfunktio. Se toimii luokan muodostajien täydentäjänä.

Tuhoaja on erityinen käyttäjän määrittelemä funktio, joka käynnistetään automaattisesti, kun luokkaolio menee viittausalueensa ulkopuolelle tai aina, kun käytetään delete-lauseketta luokkaolion osoittimeen. Tuhoajalle annetaan luokan nimi, jonka edessä on etuliitteenä tildemerkki (~). Se ei voi palauttaa arvoa eikä sille voi antaa parametreja. Koska sille ei voi määrittää parametreja, ei sitä voi ylikuormittaa. Vaikka voimme määritellä useita luokan muodostajia, voimme tehdä vain yhden tuhoajan, jota käytetään kaikille luokan olioille. Tässä on Account-luokkamme tuhoaja:

```
class Account {
public:
            Account();
            explicit Account( const char*, double=0.0 );
            Account( const Account& );
            ~Account();
    // ...
private:
          char       *_name;
          unsigned int _acct_nmbr;
          double       _balance;
};

inline
Account::~Account()
{
         delete [] _name;
         return_acct_nmbr( _acct_nmbr );
}
```

Huomaa, että emme toteuttaneet tuhoajaamme niin, että se alustaisi tietojäsentemme arvot uudelleen:

```
inline
Account::~Account()
{
        // tarpeellista
        delete [] _name;
        return_acct_nmbr( _acct_nmbr );

        // tarpeetonta
        _name = 0;
        _balance = 0.0;
        _acct_nmbr = 0;
}
```

Vaikka tämä ei ole väärin, se on tarpeetonta, koska jäseniin liittyvä muisti palautuu takaisin. Mietitäänpä yleisemmin seuraavaa luokkaa:

```
class Point3d {
public:
    // ...
private:
    float x, y, z;
};
```

Muodostaja on tarpeellinen, jotta käyttäjät voisivat alustaa kolme koordinaattijäsentä. Mutta onko tuhoaja tarpeellinen? Tuhoaja ei ole tarpeellinen tässä tapauksessa. Resurssin vapauttamista ei vaadita Point3d-luokan oliolle. Kääntäjä luo ja tuhoaa kolmen koordinaattijäsenen muistin automaattisesti jokaisen olion elinkaaren alussa ja lopussa.

Yleensä, jos luokan tietojäsenet sisältävät arvoja kuten Point3d-luokan kolme koordinaattijäsentä, ei tuhoaja ole tarpeellinen. Ei jokainen luokka vaadi tuhoajaa, vaikka tekisimme yhden tai useamman muodostajan tuolle luokalle. Tuhoajat toimivat pääasiassa hankittujen resurssien luovuttajina; jälleen kuten muistialueen vapauttamiseen tai muistin tuhoaminen, joka on varattu new-operaattorin avulla.

Tuhoajaa ei ole rajoitettu vain luovuttamaan resursseja. Yleensä tuhoaja voi tehdä minkä tahansa operaation, jonka luokan suunnittelija haluaa suorittaa olion viimeisen käytön jälkeen. Esimerkiksi yleistä tekniikkaa ohjelman suorituskyvyn instrumentointiin on määritellä Timer-luokka (Ajastin). Timer-luokan muodostaja käynnistää jonkinlaisen ohjelmakellon. Sen tuhoaja pysäyttää kellon ja näyttää jollakin tavalla tulokset. Timer-luokka voidaan sitten määritellä ehdollisesti kriittisiin ohjelmakohtiin, joihin toivomme aikaa kuten seuraavassa:

```
{
    // kriittisen ohjelmakohdan alku
#ifdef PROFILE
    Timer t;
#endif
    // kriittinen koodi
    // t:n tuhoaminen automaattisesti
    // näyttää lasketun ajan ...
}
```

Jotta voimme olla varmoja, että ymmärrämme tuhoajan toiminnan (ja muuten muodostajan myös!), käykäämme läpi esimerkki, jossa käytetään seuraavaa koodikatkelmaa:

```
(1)   #include "Account.h"
(2)   Account global( "James Joyce" );
(3)   int main()
(4)   {
(5)           Account local( "Anna Livia Plurabelle", 10000 );
(6)           Account &loc_ref = global;
(7)           Account *pact = 0;
(8)
(9)           {
(10)              Account local_too( "Stephen Hero" );
(11)              pact = new Account( "Stephen Dedalus" );
(12)          }
(13)
(14)          delete pact;
(15)    }
```

Kuinka monta muodostajaa käynnistetään? Neljä: yksi globaalille `global`-oliolle rivillä (2); yksi kummallekin kahdelle paikalliselle oliolle `local` ja `local_too` riveillä (5) ja (10) vastaavasti ja yksi keosta varatulle oliolle rivillä (11). Ei kumpikaan esittelyistä, viittaus `loc_ref`-luokkaolioon rivillä (6) eikä osoitin `pact`-luokkaolioon rivillä (7), johda muodostajan käynnistykseen. Viittaus toimii aliaksena jo muodostetulle oliolle. Tässä tapauksessa `loc_ref` toimii aliaksena `global`-oliolle. Myös osoitin osoittaa vain olioon, joka on jo muodostettu (tässä tapauksessa olio, joka on varattu keosta rivillä (11)) tai joka ei osoita mihinkään olioon (rivi (7)).

Samalla tavalla on neljä käynnistettävää tuhoajaa: yksi globaalille global-oliolle, joka on esitelty rivillä (2), yksi kummallekin paikalliselle oliolle ja yksi keko-oliolle, joka tuhotaan rivillä (14). Kuitenkin toisin kuin muodostajalla, luokkaoliolle käynnistetyn tuhoajan ilmaisuun ei liity lähdekoodin lausetta. Sen sijaan kääntäjä lisää käynnistyksen yksinkertaisesti olion viimeisen käytön jälkeen, mutta ennen vastaavan viittausalueen loppua.

Globaaleille luokkaolioille käytetään niiden muodostajia ja tuhoajia ohjelman suorituksen alustuksen ja tyhjennysvaiheiden aikana. Vaikka sellaiset globaalit oliot käyttäytyvät hyvin, jos niitä käytetään tiedostoissa, joissa ne on määritelty, tulee niiden turvallisesta ja tehokkaasta käytöstä haastava suunnitteluongelma C++:ssa, kun niihin viitataan erillisistä käännetyistä tiedostoista.[4]

Tuhoajaa ei käynnistetä, jos joko viittaus tai osoitin luokkaan siirtyy pois viittausalueelta (olio, johon viitataan, ei ole päättänyt elinkaartaan).

Kielen vartijat käyttävät sisäisesti delete-operaattoria ja poistavat osoittimen, joka ei osoita mihinkään olioon. Joten meidän ei tarvitse kirjoittaa koodia sitä varten:

```
// tarpeeton -- kääntäjä toteuttaa implisiittisesti
if ( pact != 0 ) delete pact;
```

Aina, kun yksittäinen keosta varattu olio poistetaan funktiossa, on parempi käyttää auto_ptr-luokkaoliota kuin varsinaista osoitinta (katso kohdasta 8.4 auto_ptr:n käsittely). Tämä pätee erityisesti keosta varatuille luokkaolioille, kun delete-lausekkeen käyttö epäonnistuu kuten esimerkiksi silloin, kun poikkeus heitetään. Se ei johda vain muistin loppumiseen, vaan myös siihen, että tuhoajaa ei kutsuta. Tässä on kirjoitettu uudelleen ohjelmaesimerkkimme, jossa käytetään auto_ptr-oliota (sitä on muokattu hieman, koska auto_ptr-olio ei tue merkintää, jossa se eksplisiittisesti asetetaan osoittamaan toiseen olioon, paitsi jos siihen sijoitetaan toinen auto_ptr):

```
#include <memory>
#include "Account.h"
Account global( "James Joyce" );
int main()
{
   Account local( "Anna Livia Plurabelle", 10000 );
   Account &loc_ref = global;
   auto_ptr<Account> pact( new Account( "Stephen Dedalus" ));

   {
      Account local_too( "Stephen Hero" );
   }

   // auto_ptr-olio tuhotaan täällä
```

4. Katso Jerry Schwarzin artikkeli [LIPPMAN96b] alkuperäisestä ongelmasta ja yhä vieläkin laajalle levinneestä sen ratkaisusta.

```
}
```

14.3.1 Eksplisiittinen tuhoajan käynnistys

Joissakin ohjelman tilanteissa on tarpeen käynnistää tuhoaja eksplisiittisesti tietylle luokkaolioolle. Tämä tulee eteen useimmin new-lausekkeen asemoinnin yhteydessä (katso kohdasta 8.4 sen käsittely). Katsokaamme esimerkkiä. Kun kirjoitamme

```
char *arena = new char[ sizeof Image ];
```

varaamme itse asiassa keosta muistia, jonka koko on yhtäsuuri kuin Image-tyyppisen luokkaolion koko. Kyseinen muisti on alustamatta ja täyttyy satunnaisilla bittijonoilla edellisen käytön pohjalta. Kun kirjoitamme

```
Image *ptr = new (arena) Image( "Quasimodo" );
```

ei uutta muistia varata. Sen sijaan ptr:ään sijoitetaan osoite, joka liittyy arena:an. Kuitenkin ptr:n kautta muisti tulkitaan Image-luokkaolioksi. Lisäksi, vaikka muistia ei varata, muodostajaa käytetään olemassaolevaan muistiin. Itse asiassa new-lausekkeen asemointi mahdollistaa, että voimme muodostaa luokkaolion tiettyyn, esivarattuun muistiosoitteeseen.

Kun saamme Quasimodo:n kuvan valmiiksi, haluamme käsitellä Esmereldan kuvaa samassa muistipaikassa, johon arena osoittaa. Toisaalta tiedämme täsmälleen, kuinka se tehdään:

```
Image *ptr = new (arena) Image( "Esmerelda" );
```

Ongelma on, että tämä korvaa Quasimodo-kuvamme, jota olemme muokanneet ja jonka haluamme tallentaa levylle. Tavallisesti teemme tämän Image-luokan tuhoajan kautta, mutta jos käytämme delete-operaattoria

```
// ei hyvä: poistaa muistin sekä käynnistää myös tuhoajan
delete ptr;
```

niin silloin käynnistämme lisäksi tuhoajan ja tuhoamme taustalla olevan keon muistin myös eikä se ole se, mitä halusimme. Sen sijaan voimme käynnistää eksplisiittisesti Image-tuhoajan

```
ptr->~Image();
```

ja jättää taustalla olevan muistin tulevan new-lausekkeen asemoinnin käytettäväksi.

Vaikka ptr ja arena osoittavat samaa keon muistia, ei maksa vaivaa käyttää delete-operaattoria arena:an:

```
// tuhoajaa ei käynnistetä
delete arena;
```

koska se ei johda Image-tuhoajan käynnistykseen, koska arena on char*-tyyppinen ja muista, että kääntäjä kutsuu tuhoajaa vain, jos osoitin delete-lausekkeessa osoittaa tuhoajan luokkatyyppiin.

14.3.2 Potentiaalinen ohjelmakoodivirhe

Välitön tuhoaja voi olla pahaa aavistamaton ohjelmakoodivirheen lähde, koska se lisätään jokaisen sellaisen funktion poistumiskohtaan, jossa on aktiivinen paikallinen luokkaolio. Esimerkiksi seuraavassa koodikatkelmassa

```
Account acct( "Tina Lee" );
int swt;
// ...
switch( swt ) {
case 0:
    return;
case 1:
    // tee jotain...
    return;
case 2:
    // tee jotain muuta...
    return;
// jne
}
```

pitää tuhoaja laajentaa välittömäksi ennen jokaista return-lausetta. Kun kyseessä on Account-luokan tuhoaja, sen koko on pieni ja sen useiden laajentamisten aiheuttama kuormitus on vähäinen. Jos se osoittautuu kuitenkin ongelmaksi, on ratkaisu joko esitellä tuhoaja muuksi kuin välittömäksi tai kirjoitaa ohjelmakoodi uudestaan. Eräs mahdollinen uusi ohjelmakoodiratkaisu on korvata jokaisen case-lauseen return-lause break-lauseella ja esitellä yksi paluupiste switch-rakenteen jälkeen kuten seuraavassa:

```
// kirjoitettu uudelleen ja tehty yksi paluupiste
switch( swt ) {
case 0:
    break;
case 1:
    // tee jotain...
    break;
case 2:
    // tee jotain muuta...
    break;
// jne
}

// yksi paluupiste
return;
```

Harjoitus 14.6

Kun on olemassa seuraavat luokan tietojäsenet, jossa pstring osoittaa dynaamista merkkitaulukkoa, kirjoita sopiva tuhoaja.

```
class NoName {
public:
        ~NoName();
       // ...
private:
       char  *pstring;
       int    ival;
       double dval;
};
```

Harjoitus 14.7

Päättele, onko tuhoaja tarpeen luokalle, joka valittiin kohdan 14.2 harjoituksessa 14.3. Ellei ole, perustele miksi. Tee muussa tapauksessa sen toteutus.

Harjoitus 14.8

Kuinka monen tuhoajan käynnistys tapahtuu seuraavassa koodikatkelmassa?

```
void mumble( const char *name, double balance, char acct_type )
{
        Accout acct;

        if ( ! name )
            return;

        if ( balance <= 99 )
            return;

        switch( acct_type ) {
           case 'z': return;
           case 'a':
           case 'b': return;
        }

        // ...

}
```

14.4 Luokkaoliotaulukot ja -vektorit

Luokkaoliotaulukko määritellään samalla tavalla kuin sisäisen tyypin taulukko. Esimerkiksi seuraavassa

```
Account table[ 16 ];
```

määritellään taulukko, jossa on 16 Account-oliota. Jokainen elementti alustetaan vuorollaan Account-oletusmuodostajalla. Jos haluamme, voimme antaa eksplisiittiset argumentit muodostajalle aaltosuluissa olevalla taulukon alustusluettelolla. Esimerkiksi

```
Account pooh_pals[] = { "Piglet", "Eeyore", "Tigger" };
```

määrittelee taulukon, jossa on kolme elementtiä alustettuna vuorollaan muodostajilla

```
Account( "Piglet", 0.0 );  // ensimmäinen elementti
Account( "Eeyore", 0.0 );  // toinen elementti
Account( "Tigger", 0.0 );  // kolmas elementti
```

Muodostajalle voidaan määrittää eksplisiittisesti yksittäinen argumentti kuten edellisessä esimerkissä. Jos haluamme määrittää useita argumentteja, pitää käyttää täydellistä muodostajan syntaksia. Esimerkiksi:

```
Account pooh_pals[] = {
    Account( "Piglet", 1000.0 ),
    Account( "Eeyore", 1000.0 ),
    Account( "Tigger", 1000.0 )
};
```

Jotta voisimme määrittää oletusmuodostajan taulukon alustusluetteloon, käytämme täydellistä muodostajan syntaksia, jossa on tyhjä parametriluettelo. Esimerkiksi:

```
Account pooh_pols[] = {
    Account( "Woozle", 10.0 ),
    Account( "Heffalump", 10.0 ),
    Account()
};
```

Vaihtoehtoisesti saavutamme samanarvoisen kolmen elementin taulukon kirjoittamalla

```
Account pooh_pols[3] = {
    Account( "Woozle", 10.0 ),
    Account( "Heffalump", 10.0 )
};
```

Tämä tarkoittaa, että taulukon alustusluetteloa käytetään vuorollaan jokaiselle elementille. Ne elementit, joilla ei ole eksplisiittistä muodostaja-argumenttia, alustetaan luokan oletusmuodostajalla. Ellei luokalle ole määritelty oletusmuodostajaa, pitää alustusluettelossa olla muodostaja-argumentit taulukon jokaiselle elementille.

Luokkataulukon yksittäisiä elementtejä käsitellään indeksioperaattorin kautta aivan kuten sisäisten tyyppien taulukossa. Joten esimerkiksi

```
pooh_pals[0];
```

käsittelee arvoa Piglet, kun taas

```
pooh_pals[1];
```

käsittelee arvoa Eeyore jne. Jos haluamme käsitellä tietyn taulukon elementin luokkajäseniä, yhdistämme indeksin ja jäsenen käsittelyoperaattorin. Esimerkiksi:

```
pooh_pals[1]._name != pooh_pals[2]._name();
```

Ei ole olemassa tapaa asettaa eksplisiittisiä arvoja, joilla voitaisiin alustaa keosta varatun luokkaoliotaulukon elementit. Luokalle pitää tehdä joko oletusmuodostaja tai ei muodostajia ollenkaan, jos se haluaa tukea taulukon muistinvarausta new-lausekkeen käynnistyksen kautta. Käytännössä melkein kaikilla luokilla on oletusmuodostaja.

Esittely

```
Account *pact = new Account[ 10 ];
```

luo kymmenen Account-luokkaoliotaulukon, jotka on varattu keosta. Jokainen alustetaan Account-luokan oletusmuodostajalla.

Jotta voimme vapauttaa taulukon muistin, johon `pact` osoittaa, pitää käyttää `delete`-lauseketta. Kuitenkin, jos kirjoitamme yksinkertaisesti

```
// hups: ei ihan oikein
delete pact;
```

ei se riitä, koska se ei yksilöi `pact`:ia siten, että se osoittaisi luokkaoliotaulukkoon. Vaikutus on, että Account-tuhoajaa käytetään vain ensimmäiseen elementtiin, joka ei ole kaikki, jota haluamme. Jotta tuhoajaa käytettäisiin kaikkiin taulukon elementteihin, pitää laittaa mukaan tyhjä hakasulkupari delete-operaattorin ja poistettavan olion osoitteen väliin:

```
// ok:
// ilmaisee, että pact osoittaa taulukkoon
delete [] pact;
```

Tyhjä hakasulkupari ilmaisee, että `pact` osoittaa taulukkoon. Kääntäjä hakee taulukossa olevien elementtien lukumäärän ja varmistaa, että tuhoajaa käytetään niihin jokaiseen.

14.4.1 Keosta varatun taulukon alustus

Oletusarvo on, että keosta varatun luokkaoliotaulukon alustaminen vaatii kaksi vaihetta: (1) taulukon varsinainen varaaminen muistista, jolloin taulukon oletusmuodostajaa, jos sellainen on määritelty, käytetään jokaiseen elementtiin ja (2) sen jälkeen tietyn arvon sijoittaminen jokaiseen taulukon elementtiin.

Tehdäkseen yhden alustusvaiheen, pitää ohjelmoijan määrittää alkuarvot kaikille tai osalle taulukon elementeistä ja varmistua, että oletusmuodostajaa käytetään niille elementeille, joille ei ole annettu alkuarvoa. Seuraavassa on yksi monista ohjelmaratkaisuista; siinä käytetään hyväksi new-lausekkeen asemointia.

```
#include <utility>
#include <vector>
#include <new>
#include <cstddef>
#include "Accounts.h"

typedef pair<char*,double> value_pair;

/* init_heap_array(),
 *    esittelee staattisen jäsenfunktion,
 *    varaa ja alustaa luokan oliot
 *
 *
 * init_values: alkuarvoparit taulukon elementeille
 * elem_count: elementtien lukumäärä taulukossa
 *           jos 0, on taulukon koko init_values-vektori.
 */
Account*
Account::
init_heap_array(
                vector<value_pair> &init_values,
                vector<value_pair>::size_type elem_count = 0 )
{
   vector<value_pair>::size_type
                 vec_size = init_values.size();

   if ( vec_size == 0 && elem_count == 0 )
                return 0;

   // varattavan taulukon koko on joko elem_count
   // tai, jos elem_count on 0, vektorin koko ...
   size_t elems = elem_count
                           ? elem_count : vec_size();

   // kaappaa palanen muistia taulukkoon
   char *p = new char[sizeof(Account)*elems];
```

```
    // alusta yksitellen taulukon jokainen elementti
    int offset = sizeof( Account );
    for ( int ix = 0; ix < elems; ++ix )
    {
            // siirtymä (offset) ix:teen elementtiin.
            // jos alustuspari on annettu,
            //    välitä se muodostajalle;
            // muussa tapauksessa käynnistä oletusmuodostaja

            if ( ix < vec_size )
                    new( p+offset*ix ) Account( init_values[ix].first,
                            init_values[ix].second );
            else new( p+offset*ix ) Account;
    }

    // ok: elementit varattu ja alustettu;
    //     palauta osoitin ensimmäiseen elementtiin
    return (Account*)p;
}
```

Tässä niksi on “esivarata” palanen muistia, joka on riittävä pyydetylle luokkataulukolle. Varaamme sen vapaasta muistista ja vältämme siten oletusmuodostajan käynnistämisen jokaiselle taulukon elementille. Sen tekee seuraava lause:

```
char *p = new char[sizeof(Account)*elems];
```

Seuraavaksi ohjelma käy läpi muistipalaa siirtäen `p`:tä seuraavaan Account-elementin osoitteeseen ja käynnistää kaksiparametrisen muodostajan, jos alustuspari on annettu, tai sitten oletusmuodostajan:

```
for ( int ix = 0; ix < elems; ++ix ) {
    if ( ix < vec_size )
        new( p+offset*ix ) Account( init_values[ix].first,
                        init_values[ix].second );
    else new( p+offset*ix ) Account;
}
```

Kuten näimme kohdassa 14.3, new-operaattorin asemointi mahdollistaa sen, että voimme käyttää luokan muodostajaa esivarattuun muistialueeseen. Tässä tapauksessa käytämme new-operaattorin asemointia ja Account-luokan muodostajaa jokaiselle esivaratulle taulukon elementille vuorollaan. Koska olemme korvanneet tavanomaisen muistinvarausmekanismin generoimalla oman alustetun, keosta varatun luokkataulukon, pitää järjestää tuki myös muistin vapauttamiselle. Tavanomaisen delete-operaattorin käyttö ei toimi:

```
delete [] ps;
```

Miksi? ps:ää (oletamme, että se on alustettu kutsumalla init_heap_array():tä) ei ole varattu tavallisella taulukon new-operaattorilla, joten siihen liittyvien elementtien lukumäärä ei ole tiedossa. Meidän pitää tehdä työ itse:

```
void
Account::
dealloc_heap_array( Account *ps, size_t elems )
{
   for ( int ix = 0; ix < elems; ++ix )
       ps[ix].Account::~Account();

   delete [] reinterpret_cast<char*>(ps);
}
```

Jos muistat, käytimme alustusfunktiossa osoitinaritmetiikkaa jokaisen elementin käsittelyyn

```
new( p+offset*ix ) Account;
```

kun taas tässä käsittelemme jokaista elementtiä indeksoimalla ps:ää:

```
ps[ix].Account::~Account();
```

Ero on, että vaikka sekä ps että p osoittavat samaan muistialueeseen, ps on esitelty osoittimena Account-luokkaolioon, kun taas p on esitelty osoittimena char-tyyppiin. Jos indeksoisimme kohtaan p, se johtaisi ix:nteen taulukon tavuun eikä ix:nteen Account-luokkaolioon. Koska vastaava tyyppi ei kelpaa p:lle, pitää osoitinaritmetiikka ohjelmoida itse.

Esittelemme molemmat funktiot luokan staattisiksi jäseniksi:

```
typedef pair<char*,double> value_pair;

class Account {
public:
      // ...
   static Account* init_heap_array(
               vector<value_pair> &init_values,
               vector<value_pair>::size_type elem_count = 0 );
   static void dealloc_heap_array( Account*, size_t );
   // ...
};
```

14.4.2 Luokkaolioiden vektori

Kun määrittelemme viiden luokkaolion vektorin kuten

```
vector< Point > vec( 5 );
```

tapahtuu elementtien alustus seuraavasti[5]:

1. Luodaan tilapäinen kyseessä olevan luokan tyyppinen olio. Luokan oletusmuodostajaa käytetään sen luomiseen.
2. Käytetään vektorin jokaiseen elementtiin vuorollaan kopiointimuodostajaa, joka alus-

taa jokaisen luokkaolion tilapäisen luokkaolion kopiolla.

3. Tilapäinen luokkaolio tuhotaan.

Vaikka lopullinen tulos on sama kuin määrittelisi viiden luokkaolion taulukon kuten

```
Point pa[ 5 ];
```

maksaa vektorin alustaminen enemmän: (1) tilapäisen olion muodostaminen ja tuhoaminen tietysti ja (2) kopiointimuodostajat tahtovat olla laskennallisesti monimutkaisempia kuin oletusmuodostajat.

Yleisenä suunnittelusääntönä luokkaoliovektori sopii parhaiten vain elementtien lisäämiseen; tarkoittaa, että määrittelemme tyhjän vektorin. Jos olemme laskeneet etukäteen lisättävien elementtien lukumäärän tai meillä on hyvä arvaus sen koosta, varaamme siihen liittyvän muistin. Sitten etenemme elementtien lisäykseen. Esimerkiksi:

```
vector< Point > cvs; // tyhjä
int cv_cnt = calc_control_vertices();

// varaa muistia niin, että siihen mahtuu cv_cnt Point oliota
// cvs on yhä tyhjä ...
cvs.reserve( cv_cnt );

// avaa tiedosto ja valmistaudu käymään se läpi
ifstream infile( "spriteModel" );
istream_iterator<Point> cvfile( infile ),eos;

// ok, lisää nyt elementit
copy( cvfile, eos, inserter( cvs, cvs.begin() ));
```

(copy(), lisääjäluokka ja istream_iterator on käsitelty luvussa 12.) Lista- ja pakkaolioiden määrittely noudattelee samaa menettelyä kuin vektoriolioidenkin. Luokkaolioiden lisäys kuhunkin säiliötyyppiin on toteutettu kopiointimuodostajaa käyttäen.

Harjoitus 14.9

Mitkä seuraavista ovat virheellisiä, vai onko yksikään? Korjaa jokainen ilmentymä, jota pidät virheellisenä.

```
(a) Account *parray[10] = new Account[10];
(b) Account iA[1024] = {
   "Nhi", "Le", "Jon", "Mike", "Greg", "Brent", "Hank"
   "Roy", "Elena" };
(c) string *ps=string[5]("Tina","Tim","Chyuan","Mira","Mike");
(d) string as[] = *ps;
```

5. Tunnusomainen muodostaja on seuraavassa. Kopiointimuodostaja käyttää arvoa jokaiselle elementille vuorollaan. Kun annetaan luokkaolio toisena argumenttina, on tilapäisen olion luonti tarpeeton.

```
explicit vector( size_type n, const T& value=T(),const Allocator&=Allocator());
```

Harjoitus 14.10

Missä seuraavissa tilanteissa on sopivampaa, vai onko yhdessäkään, että staattinen taulukko kuten Account pA[10] on dynaaminen taulukko tai vektori? Perustele, miksi.

(a) Tarvitaan 256 elementin kokoelma Color-luokkaolioiden tallentamiseen funktiossa nimeltään Lut(). Arvot ovat vakioita.

(b) Tarvitaan tuntematon määrä Account-elementtejä. Jokaisen tilin tieto on tallennettu luettavaan tiedostoon.

(c) Generoidaan kokoelma elem_size-kokoisia merkkijonoja ja välitetään takaisin tekstinkäsittelijälle funktiolla gen_words(elem_size).

Harjoitus 14.11

Potentiaalinen virheen paikka dynaamisten luokkataulukoiden käytössä on unohtaa hakasulut, jotka ilmaisevat, että osoitin osoittaa taulukkoon; tarkoittaa, että jos kirjoitamme

```
// hups: ei tarkistusta, jos parray osoittaakin taulukkoa
delete parray;
```

kirjoitamme sen sijaan

```
// ok: hae taulukon koko, jota parray osoittaa
delete [] parray;
```

Hakasulkupari saa aikaan sen, että kääntäjä hakee taulukon koon. Tuhoajaa käytetään sitten size kertaa taulukon elementeille. Muussa tapauksessa tuhotaan vain yksi elementti. Kaikki varattu muisti palautetaan vapaavarastoon joka tapauksessa.

Alkuperäisen kielen suunnittelussa käytettiin valtavasti aikaa keskusteluun, pitäisikö vaatia hakasulkuparia haun alkuun saamiseksi vai toisaalta säilyttää alkuperäinen kielen vaatimus ohjelmoijalle laittaa taulukon eksplisiittinen koko hakasulkuihin:

```
// alkuperäinen kielen suunnittelu vaati eksplisiittistä kokoa
delete [10] parray;
```

Mitä luulet, miksi kieltä muutettiin niin, että käyttäjän ei tarvitse laittaa hakasulkuihin taulukon eksplisiittistä kokoa, mikä vaatii muistia ja koon hakemista, vaan jättää pelkän hakasulkuparin delete-lausekkeen yhteyteen, mikä vaatii, että toteutus muistaa, että osoitin osoittaa joko yhtä oliota tai taulukkoa? Kuinka sinä olisit suunnitellut kielen tässä tapauksessa?

14.5 Jäsenen alustusluettelo

Muokatkaamme Account-luokkaamme esittelemällä uudelleen _name-jäsen string-tyyppisenä:

```
#include <string>
class Account {
public:
        // ...
private:
        unsigned int _acct_nmbr;
        double       _balance;
        string       _name;
};
```

Muodostajiamme pitää muokata myös. Tähän liittyy kaksi asiaa: (1) alkuperäisen rajapinnan yhteensopivuuden säilyttäminen samalla, kun vaihdamme uuteen tyyppiin ja (2) luokkaolioiden jäsenten alustaminen niihin liittyvillä muodostajilla.

Alkuperäinen kaksiparametrinen Account-muodostaja

```
Account( const char*, double = 0.0 );
```

on riittämätön uudelle string-luokkatyypillemme. Esimerkiksi

```
string new_client( "Steve Hall" );
Account new_acct( new_client, 25000 );
```

epäonnistuu, koska ei ole olemassa implisiittistä konversiota string-oliosta char*-tyypiksi. Kun kirjoitamme

```
Account new_acct( new_client.c_str(), 25000 );
```

se toimii, mutta osoittautuu todennäköisesti hämmentäväksi käyttäjille. Eräs ratkaisu on lisätä uusi muodostaja muotoa

```
Account( string, double = 0.0 );
```

Nyt, kun kirjoitamme

```
Account new_acct( new_client, 25000 );
```

tämä ilmentymä käynnistyy, kun taas vanhempi koodi kuten

```
Account *open_new_account( const char *nm )
{
        Account *pact = new Account( nm );
        // ...
        return pact;
}
```

jatkaa alkuperäisen kaksiparametrisen ilmentymän käynnistämistä.

Koska string-luokassa on konversio char*-tyypistä string-olioksi (luokkakonversioita käsitellään seuraavassa luvussa), voimme myös yksinkertaisesti korvata alkuperäisen kaksiparametrisen muodostajan uudella ilmentymällä, joka saa ensimmäisen parametrin string-tyyppisenä. Tässä tapauksessa, kun kirjoitamme

```
Account myAcct( "Tinkerbell" );
```

konvertoidaan "Tinkerbell" tilapäiseksi string-olioksi. Tuo olio välitetään sitten kaksiparametriselle muodostajalle, joka saa ensimmäisen parametrin string-tyyppisenä.

Suunnittelussa tasapainottelevat Account-muodostajien lisääntyminen vastaan hieman tehottomampi char*-argumenttien käsittely, joka johtuu string-olion tilapäiseksi luomisesta. Suunnitteluvalintamme on tehdä kaksi versiota kaksiparametrisesta muodostajasta. Uudistetut Account-muodostajamme ovat seuraavassa:

```
#include <string>

class Account {
public:
        Account();
        Account( const char*, double=0.0 );
        Account( const string&, double=0.0 );
        Account( const Account& );
        // ...
private:
        // ...
};
```

Seuraava asia on, kuinka luokan tietojäsen, johon liittyy muodostajia, alustetaan kunnolla. Tämä jakautuu seuraavaan kolmeen alikategoriaan:

1. Kuinka käynnistämme sen oletusmuodostajan? Se pitää tehdä Account-oletusmuodostajalla.
2. Kuinka käynnistämme sen kopiointimuodostajan? Se pitää tehdä Account-kopiointimuodostajalla, joka saa ensimmäisenä parametrinaan string-tyypin.
3. Yleisemmin, kuinka välitämme argumentteja jäsenluokkaolion muodostajalle? Se pitää tehdä kaksiparametrisella Account-muodostajalla, joka saa char*-tyypin ensimmäisenä parametrinaan.

Ratkaisu on jäsenen alustusluettelo (esitelty lyhyesti kohdassa 14.2). Luokan tietojäsenet voidaan alustaa eksplisiittisesti jäsenen alustusluettelolla, joka on pilkuin eroteltu luettelo jäsen/nimi-argumenttipareja. Tässä on esimerkkinä uudelleentoteuttamamme kaksiparametrinen muodostaja, jossa käytetään jäsenen alustusluetteloa (muista, että _name on nyt string-tyyppinen jäsenluokkaolio):

```
inline Account::
Account( const char* name, double opening_bal )
        : _name( name ), _balance( opening_bal )
{
        _acct_nmbr = get_unique_acct_nmbr();
}
```

Jäsenen alustusluettelo tulee muodostajan nimen jälkeen ja se ilmaistaan kaksoispisteellä. Jäsenen nimi määritetään, minkä jälkeen tulevat alkuarvot sulkujen sisällä; syntaksi, joka muistuttaa funktion kutsua. Jos jäsen on luokkaolio, alkuarvoista tulee argumentteja, jotka välitetään sopivalle muodostajalle, jota sitten käytetään jäsenluokkaolioon. Esimerkissämme `name` välitetään string-muodostajalle, jota käytetään `_name`:een. `_balance` alustetaan `opening_bal`-parametrilla.

Samalla tavalla tässä on toinen kaksiparametrinen Account-muodostajamme:

```
inline Account::
Account( const string& name, double opening_bal )
        : _name( name ), _balance( opening_bal )
{
        _acct_nmbr = get_unique_acct_nmbr();
}
```

Tässä tapauksessa käynnistetään string-kopiointimuodostaja, joka alustaa `_name`-jäsenluokkaolion merkkijonolla, joka on parametrissa `name`.

Vasta-alkaneiden C++-ohjelmoijien yleinen kysymys koskee eroa alustusluettelon käytön ja tietojäsenten sijoituksen välillä muodostajan rungossa. Esimerkiksi, mitä eroa on

```
inline Account::
Account( const char *name, double opening_bal )
        : _name( name ), _balance( opening_bal )
{
        _acct_nmbr = get_unique_acct_nmbr();
}
```

ja

```
inline Account::
Account( const char *name, double opening_bal )
{
        _name = name;
        _balance = opening_bal;
        _acct_nmbr = get_unique_acct_nmbr();
}
```

välillä?

Molempien toteutusten lopputulos on samanlainen. Molempien muodostajien käynnistyksen lopussa on kolmella jäsenellä samat arvot. Ero on, että vain jäsenen alustusluettelolla saadaan luokan tietojäsenet alustettua. Muodostajan rungossa tietojäsenen arvon asetus on sijoitus. Tämän erottelun merkitys riippuu tietojäsenen tyypistä.

Käsitteellisesti on tärkeää ajatella muodostajan suoritusta kahtena vaiheena: (1) joko implisiittisenä tai eksplisiittisenä alustusvaiheena ja (2) yleisenä suoritusvaiheena. Suoritusvaihe koostuu kaikista muodostajan rungon lauseista. Kaikki tietojäsenten asetukset suorituksen vaiheen aikana luokitellaan sijoituksiksi eikä alustamisiksi. Tämän eron selkeänä pitämisen epäonnistuminen on yleinen ohjelmavirheiden ja ohjelman tehottomuuden lähde.

Alustusvaihe voi olla joko implisiittinen tai eksplisiittinen riippuen siitä, onko jäsenen alustusluettelo mukana. Implisiittinen alustusvaihe käynnistää esittelyjärjestyksessä kaikki kantaluokan oletusmuodostajat, sitten kaikki jäsenluokkaolioiden oletusmuodostajat. (Katsomme kantaluokkia luvussa 17, kun käsittelemme oliosuuntautunutta ohjelmointia.) Kun esimerkiksi kirjoitamme

```
inline Account::
Account()
{
        _name = "";
        _balance = 0.0;
        _acct_nmbr = 0;
}
```

on alustusvaihe implisiittinen. Ennen kuin muodostajamme runko on suoritettu, käynnistetään string-oletusmuodostaja, joka liittyy _name:en Tämä tarkoittaa, että null-merkkijonon sijoitus _name:en on tarpeeton.

Luokkaolioille alustamisen ja sijoituksen välinen ero on merkittävä. Luokkaolion jäsen tulisi aina alustaa jäsenen alustusluettelolla sen sijaan, että siihen sijoitettaisiin muodostajan rungossa. Oikeampi toteutus Account-oletusmuodostajasta on seuraavassa:

```
inline Account::
Account() : _name( string() )
{
        _balance = 0.0;
        _acct_nmbr = 0;
}
```

On oikeammin, että olemme poistaneet tarpeettoman sijoituksen _name:en muodostajassa. Oletusmuodostajan eksplisiittinen käynnistys on kuitenkin tarpeeton. Seuraavassa on tiiviimpi, mutta silti samanarvoinen toteutus:

```
inline Account::
Account()
{
        _balance = 0.0;
        _acct_nmbr = 0;
}
```

Tämä jättää silti meille kysymyksen, joka koskee kahden tietojäsenen alustamista, jotka on esitelty sisäisillä tyypeillä. Onko esimerkiksi sama asia, alustammeko _balance:n jäsenen alustusluettelolla vai muodostajan rungossa? Vastaus on ei. Luokattoman tietojäsenen alustaminen tai sijoitus kahta poikkeusta lukuunottamatta on samanarvoinen niin tuloksiltaan kuin suorituskyvyltään. Kuten sanottua, parempi toteutuksemme käyttää jäsenen alustusluetteloa:

```
// parempi alustustyyli
inline Account::
Account() : _balance( 0.0 ), _acct_nmbr( 0 )
{}
```

Ne kaksi poikkeusta ovat minkä tahansa tyyppiset const- ja viittaustietojäsenet. const- ja viittaustietojäsenet pitää aina alustaa jäsenen alustusluettelolla; muussa tapauksessa saadaan aikaan käännöksenaikainen virhe. Esimerkiksi seuraava muodostajan toteutus johtaa käännösvirheeseen:

```
class ConstRef {
public:
   ConstRef( int ii );
private:
   int i;
   const int ci;
   int &ri;
};

ConstRef::
ConstRef( int ii )
{ // sijoitus
    i = ii;   // ok
    ci = ii;  // virhe: ei voi sijoittaa const-tyyppiin
    ri = i;   // virhe: ri on alustamatta
}
```

Jossain vaiheessa, kun muodostajan runko aloittaa suorituksensa, pitää kaikkien const- ja viittaustietojäsenten alustusten jo olla tapahtuneita. Se voidaan tehdä määrittämällä niille jäsenten alustusluettelot. Oikea alustus on seuraavassa:

```
// ok: alusta viittaus ja const
ConstRef::
ConstRef( int ii )
      : ci( ii ), ri( i )
{ i = ii; }
```

Jokainen jäsen voidaan nimetä vain kerran jäsenen alustusluettelossa. Alustuksen järjestystä ei päätellä nimien järjestyksestä, vaan jäsenten esittelyjärjestyksestä. Esimerkiksi seuraavan Account-tietojäsenten esittelyjärjestyksen

```
class Account {
public:
        // ...
private:
        unsigned int _acct_nmbr;
        double       _balance;
        string       _name;
};
```

alustusjärjestys seuraavassa oletusmuodostajan toteutuksessa

```
inline Account::
Account() : _name( string() ), _balance( 0.0 ), _acct_nmbr( 0 )
{}
```

on _acct_nmbr, _balance ja sitten _name. Kuitenkin jäsenet, jotka on nimetty alustusluettelossa (sekä implisiittisesti alustetut luokkaolion jäsenet), alustetaan aina ennen jäseniin sijoituksia muodostajan rungossa. Esimerkiksi seuraavassa muodostajassa

```
inline Account::
Account( const char *name, double bal )
        : _name( name ), _balance( bal )
{
        _acct_nmbr = get_unique_acct_nmbr();
}
```

on alustuksen järjestys _balance, _name ja sitten _acct_nmbr.

Tämä ilmeinen säännöttömyys alustamisen järjestyksen ja alustusluettelon järjestyksen välillä voi johtaa seuraavaan vaikeasti löydettävään virheeseen, kun luokan jäsentä käytetään toisen alustamiseen:

```
class X {
    int i;
    int j;
public:
    // hups!  huomaatko ongelman?
    X( int val )
        : j( val ), i( j )
        {}
    // ...
};
```

Vaikka näyttääkin siltä, että j on alustettu val:illa ennen kuin sitä on käytetty i:n alustamiseen, itse asiassa i on alustettu ensin ja sen vuoksi alustetaan vielä alustamattomalla j:llä. Suosituksemme on sijoittaa aina jäsenen alustus toisella (jos tuntuu todella siltä, että se on välttämätöntä) muodostajan runkoon, kuten seuraavassa:

```
// parempi käytäntö
X::X( int val ) : i( val ) { j = i; }
```

Harjoitus 14.12

Mikä on virheellistä seuraavan muodostajan määrittelyssä, vai onko mitään? Kuinka korjaisit tunnistamasi virheen?

```
(a) Word::Word( char *ps, int count = 1 )
       : _ps( new char[strlen(ps)+1] ),
         _count( count )
    {
       if ( ps )
          strcpy( _ps, ps );
       else {
          _ps = 0;
          _count = 0;
       }
    }

(b) class CL1 {
    public:
       CL1() { c.real(0.0); c.imag(0.0); s = "not set"; }
       // ...
    private:
       complex<double> c;
       string s;
    };

(c) class CL2 {
    public:
       CL2( map<string,location> *pmap, string key )
          : _text( key ), _loc( (*pmap)[key] ) {}
       // ...
    private:
       location _loc;
       string   _text;
    };
```

14.6 Jäsenittäinen alustus 

Yhden luokkaolion alustamista toisella luokkansa oliolla kuten seuraavassa

```
Account oldAcct( "Anna Livia Plurabelle" );
Account newAcct( oldAcct );
```

sanotaan *jäsenittäiseksi oletusalustukseksi* (*default memberwise initialization*). *Oletus,* koska se tapahtuu automaattisesti, vaikka emme tekisi eksplisiittistä muodostajaa. *Jäsenittäinen,* koska alustamisen yksikkö on yksittäinen ei-staattinen tietojäsen eikä biteittäinen koko luokkaolion kopiointi.

Yksinkertaisin käsitemalli jäsenittäisestä alustamisesta on ajatella, että kääntäjä generoi erityisen luokan kopiointimuodostajan sisäisesti. Kopiointimuodostajassa jokainen ei-staattinen tietojäsen alustetaan vuorollaan esittelyjärjestyksessä. Esimerkiksi seuraavassa ensimmäisessä Account-luokan määrittelyssä

```
class Account {
public:
        // ...
private:
        char       *_name;
        unsigned int _acct_nmbr;
        double      _balance;
};
```

voidaan oletusarvoisen Account-kopiointimuodostajan ajatella määritellyn seuraavasti:

```
inline Account::
Account( const Account &rhs )
{
        _name = rhs._name;
        _acct_nmbr = rhs._acct_nmbr;
        _balance = rhs._balance;
}
```

Luokkaolion alustaminen toisella luokkansa oliolla tapahtuu seuraavissa ohjelmatilanteissa:

1. Yhden luokkaolion eksplisiittisessä alustamisessa toisella; esimerkiksi:

   ```
   Account newAcct( oldAcct );
   ```

2. Luokkaolion välittämisessä argumenttina funktioon; esimerkiksi:

   ```
   extern bool cash_on_hand( Account acct );

   if ( cash_on_hand( oldAcct ))
      // ...
   ```

3. Luokkaolion välittämisessä funktion paluuarvona; esimerkiksi:

   ```
   extern Account
     consolidate_accts( const vector< Account >& )
   ```

```
{
    Account final_acct;
    // laske rahat ...
    return final_acct;
}
```

4. Kun määritellään muu kuin tyhjä jonosäiliötyyppi; esimerkiksi:

```
// viisi string-kopiointimuodostajaa käynnistyy
vector< string > svec( 5 );
```

(Tässä esimerkissä luodaan yksi tilapäinen käyttäen string-oletusmuodostajaa ja sitten tämä tilapäinen kopioidaan vuorollaan vektorin viiteen elementtiin käyttäen string-kopiointimuodostajaa.)

5. Kun lisätään luokkaolio säiliötyyppiin; esimerkiksi:

```
svec.push_back( string( "pooh" ));
```

Useimmille käytännön elämän luokkamäärittelyille on oletusarvoinen jäsenittäinen alustus riittävä luokan turvalliselle ja oikealle käytölle. Tämä tapahtuu yleisimmin, kun luokan tietojäsen on osoitin, joka osoittaa kekomuistiin, joka on poistettu luokan tuhoajalla kuten _name Account-luokassamme.

Jäsenittäisen oletusalustamisen jälkeen sekä newAcct._name että oldAcct._name osoittavat samaan C-tyyliseen merkkijonoon. Jos oldAcct menee viittausalueen ulkopuolelle ja Account-tuhoajaa käytetään siihen, osoittaa newAcct._name nyt poistettua muistia. Vaihtoehtoisesti, jos newAcct muokkaa _name:n osoittamaa merkkijonoa, se muokkaa sitä myös oldAcct:ille. Tällaiset osoitusvirheet voivat olla vaikeita jäljittää.

Eräs ratkaisu osoittimen “alias”-ongelmaan on varata toinen kopio merkkijonosta ja alustaa newAcct._name sen osoitteella. Jotta se voitaisiin tehdä, pitää jäsenittäinen oletusalustus korvata Account-luokassamme. Teemme sen luomalla eksplisiittisen ilmentymän kopiointimuodostajasta, joka toteuttaa luokan oikean alustuksen.

Luokan sisäinen tulkinta voi myös kumota jäsenittäisen oletusalustuksen. Kuten selitimme aikaisemmin, ei kahdella Account-luokan oliolla tulisi koskaan olla samaa tilinumeroa. Taataksemme tämän, pitää Account-luokkamme jäsenittäinen oletusalustus korvata. Tässä on kopiointimuodostajamme, joka ratkaisee molemmat ongelmat:

```
inline Account::
Account( const Account &rhs )
{
        // käsittele osoittimen alias-ongelma
        _name = new char[ strlen(rhs._name)+1 ];
        strcpy( _name, rhs._name );

        // käsittele yksilöllisen tilinumeron ongelma
        _acct_nmbr = get_unique_acct_nmbr();
```

```
        // ok: tämä jäsenittäinen kopiointi toimii …
        _balance = rhs._balance;
}
```

Monissa tapauksissa ratkaisun vaikein asia on se, että yksinkertaisesti havaitaan, että sellainen tarvitaan.

Vaihtoehtona kopiointimuodostajan teolle on jäsenittäisen alustamisen kieltäminen kokonaan. Tämä voidaan tehdä seuraavilla vaiheilla:

1. Esittele kopiointimuodostaja yksityiseksi. Tämä estää jäsenittäisen alustamisen tapahtumisen missä tahansa ohjelmassa paitsi jäsenfunktioissa ja luokan ystävissä.
2. Estä jäsenittäinen alustus luokan ystävissä ja jäsenfunktioissa jättämällä tahallaan määrittelyn tekeminen pois (tarvitsemme silti vaiheen 1 esittelyn). Kieli ei salli meidän estää jäsenfunktion tai ystäväluokan käsitellä yksityisiä luokan jäseniä. Kuitenkin, jos jätämme määrittelyn pois, vaikka käännösjärjestelmä sen sallii, jokainen yritys käynnistää kopiointimuodostaja generoi linkitysvaiheen aikana virheen, koska määrittelyä ei löydy sen ratkaisemiseen.

Jotta voisimme esimerkiksi kieltää jäsenittäisen alustamisen Account-luokalta, esittelemme luokan seuraavasti:

```
class Account {
public:
        Account();
        Account( const char*, double=0.0 );

        // ...
private:
        Account( const Account& );
        // ...
};
```

14.6.1 Luokkaolion jäsenen alustus

Mitä tapahtuu, kun korvaamme C-tyylisen _name-merkkijonomme esittelyn string-luokkatyypin _name-esittelyllä? Kuinka se vaikuttaa jäsenittäiseen oletusalustukseen? Kuinka eksplisiittistä kopiointimuodostajaamme pitää muuttaa? Katsomme näitä jokaista kysymystä vuorollaan tässä alikohdassa.

Jäsenittäinen oletusalustus tutkii jokaisen jäsenen vuorollaan. Jos jäsen on sisäistä tai yhdistettyä tietotyyppiä, jäsen jäseneen -alustus etenee heti. Esimerkiksi alkuperäisessä Account-luokkamme määrittelyssä _name on osoitin ja siksi alustetaan suoraan

```
newAcct._name = oldAcct._name;
```

Luokkaolioiden jäsenet sen sijaan käsitellään eri tavalla. Kun kirjoitamme

```
Account newAcct( oldAcct );
```

nämä kaksi oliota tunnistetaan Account-luokan olioiksi. Jos Account-luokassa on eksplisiittinen kopiointimuodostaja, se käynnistetään alustuksen loppuunsaattamiseksi; muussa tapauksessa käytetään jäsenittäistä oletusalustusta.

Samalla tavalla, kun luokkaolion jäsen havaitaan, käytetään samaa prosessia rekursiivisesti. Onko luokalla eksplisiittistä kopiointimuodostajaa? Jos sillä on, käynnistetään tuo ilmentymä luokkaolion jäsenen alustamiseen. Muussa tapauksessa käytetään jäsenittäistä oletusalustusta luokkaolion jäseneen. Jos kaikki tuon luokan jäsenet ovat sisäisiä tai yhdistettyjä tietotyyppejä, alustetaan jokainen vuorollaan, jolloin saadaan loppuun luokkaolion jäsenen alustus. Muussa tapauksessa, jos yksi tai useampi noista jäsenistä on itse luokkaolion jäseniä, käytetään prosessia rekursiivisesti, kunnes jokainen sisäinen ja yhdistetty tietotyyppi on käsitelty.

Esimerkissämme string-luokalla on eksplisiittinen kopiointimuodostaja. _name alustetaan tuon ilmentymän käynnistyksen kautta. Oletusarvoisen Account-luokan kopiointimuodostajan voidaan nyt ajatella olevan määritelty seuraavasti:

```
inline Account::
Account( const Account &rhs )
{
        _acct_nmbr = rhs._acct_nmbr;
        _balance = rhs._balance;

        // C++-pseudokoodia
        // kuvaa luokkaolion jäsenen
        // kopiointimuodostajan käynnistys
        _name.string::string( rhs._name );
}
```

Account-luokan jäsenittäinen oletusalustus käsittelee nyt _name:n varaamisen ja vapauttamisen oikein, mutta silti yhä kopioi tilinumeron virheellisesti, ja siksi on tehtävä eksplisiittinen kopiointimuodostaja. Seuraava ei ole ihan oikein. Huomaatko, miksi?

```
// ei ihan oikein ...
inline Account::
Account( const Account &rhs )
{
        _name = rhs._name;
        _balance = rhs._balance;
        _acct_nmbr = get_unique_acct_nmbr();
}
```

Toteutus ei ole oikein, koska epäonnistumme erottamaan alustamisen ja sijoituksen toisistaan. Tuloksena on, että sen sijaan, että käynnistäisimme string-luokan kopiointimuodostajan, käynnistämmekin string-luokan oletusmuodostajan implisiittisessä alustusvaiheessa ja string-luokan kopioinnin sijoitusoperaattorin muodostajan rungossa. Korjaus on yksinkertainen:

```
inline Account::
Account( const Account &rhs )
        : _name( rhs._name )
{
    _balance = rhs._balance;
    _acct_nmbr = get_unique_acct_nmbr();
}
```

Jälleen kerran, todellinen työ on havaita, että meidän pitää käyttää korjausta ensi tilassa. (Molemmat toteutukset johtavat siihen, että `_name` sisältää `rhs._name`:n arvon. Ensimmäinen toteutus vaatii yksinkertaisesti kaksinkertaisen työn.) Yleinen peukalosääntö on alustaa kaikki jäsenluokkaoliot jäsenen alustusluettelolla.

Harjoitus 14.13

Mikä luokkamäärittely tarvitsee todennäköisesti kopiointimuodostajan?

(a) Point3w-esitys, joka sisältää neljä float-jäsentä.

(b) Matrix-luokka, jossa varsinainen matriisi varataan dynaamisesti muodostajassa ja se poistetaan tuhoajassa.

(c) Payroll-luokka, jossa jokainen olio varustetaan yksilöllisellä tunnisteella (ID).

(d) Word-luokka, joka sisältää string- ja vector-olion rivi- ja sarakeparina.

Harjoitus 14.14

Toteuta seuraaville luokille kopiointimuodostaja sekä oletusmuodostaja ja tuhoaja.

```
(a) class BinStrTreeNode {
   public:
      // ...
   private:
      string _value;
      int    _count;
      BinStrTreeNode *_leftchild;
      BinStrTreeNode *_rightchild;
   };

(b) class BinStrTree {
   public:
      // ...
   private:
      BinStrTreeNode *_root;
   };

(c) class iMatrix {
   public:
      // ...
   private:
      int  _rows;
      int  _cols;
      int *_matrix;
   };

(d) class theBigMix {
   public:
      // …
   private:
      BinStrTree     _bst;
      iMatrix        _im;
      string         _name;
      vector<float> *_pvec;
   };
```

Harjoitus 14.15

Mieti, onko kohdan 14.2 harjoituksessa 14.3 luokan muodostaja tarpeellinen. Ellei ole, perustele, miksi. Tee muussa tapauksessa sen toteutus.

Harjoitus 14.16

Yksilöi jokainen jäsenittäisen alustuksen ilmentymä tässä koodikatkelmassa:

```
Point global;

Point foo_bar( Point arg )
{
   Point local = arg;
   Point *heap = new Point( global );
   *heap = local;
   Point pa[ 4 ] = { local, *heap };
   return *heap;
}
```

14.7 Jäsenittäinen sijoitus ◆

Luokkaolion sijoitusta toiseen luokkansa olioon sanotaan *jäsenittäiseksi oletussijoitukseksi* (*default memberwise assignment*). Mekanismit ovat oleellisesti samat kuin jäsenittäisessä oletusalustuksessa, mutta tässä käytetään hyväksi kopiointimuodostajan implisiittistä kopioinnin sijoitusoperaattoria. Esimerkiksi

```
newAcct = oldAcct;
```

sijoittaa oletusarvoisesti newAcct:in jokaiseen ei-staattiseen jäseneen vuorollaan vastaavan oldAcct:in jäsenen arvon. Jälleen, teoriassa se on sama asia, kuin jos kääntäjä generoisi seuraavan kopioinnin sijoitusoperaattorin:

```
inline Account&
Account::
operator=( const Account &rhs )
{
        _name = rhs._name;
        _balance = rhs._balance;
        _acct_nmbr = rhs._acct_nmbr;
}
```

Yleensä, jos jäsenittäinen oletusalustus ei ole sopiva luokalle, ei jäsenittäinen oletussijoituskaan ole sopiva. Esimerkiksi alkuperäisessä Account-luokan määrityksessä, jossa _name on esitelty char*-tyyppiseksi, ovat _name:n ja _acct_nmbr:n jäsenittäiset sijoitukset sopimattomia.

Korvaamme jäsenittäisen sijoituksen tekemällä eksplisiittisen ilmentymän kopioinnin sijoitusoperaattorista, jolla toteutamme oikean luokan kopioinnin. Kopioinnin sijoitusoperaattorin yleinen muoto on seuraava:

```
// kopioinnin sijoitusoperaattorin yleinen muoto
className&
className::
operator=( const className &rhs )
{
```

```
        // vartioidaan itseensä sijoitusta
        if ( this != &rhs )
        {
                // luokan kopiointiasiat tulevat tähän
        }

        // palauta olio sijoitettuna
        return *this;
}
```

jossa ehdollinen testaus

```
if ( this != &rhs )
```

estää luokkaolion sijoittamisen itseensä. Tämä on etenkin epäsopivaa kopioinnin sijoitusoperaattorissa, joka vapauttaa ensin nykyiseen luokkaan liittyvän resurssin sijoittaakseen sen kopioitavaan luokkaan. Mietitäänpä Account-luokan kopioinnin sijoitusoperaattoria:

```
Account&
Account::
operator=( const Account &rhs )
{
        // vartioidaan itseensä sijoittamista
        if ( this != &rhs )
        {
                delete [] _name;
                _name = new char[strlen(rhs._name)+1];
                strcpy( _name,rhs._name );
                _balance = rhs._balance;
            _acct_nmbr = rhs._acct_nmbr;
        }
        return *this;
}
```

Kun luokkaolio sijoitetaan toiseen luokkansa olioon kuten tässä

```
newAcct = oldAcct;
```

tapahtuu seuraavat vaiheet:

1. Luokka tutkitaan, jotta voidaan päätellä, onko eksplisiittistä kopioinnin sijoitusoperaattoria tehty.
2. Jos on tehty, tutkitaan sen käsittelytaso: voidaanko se käynnistää tässä ohjelmanosassa.
3. Jos siihen ei päästä käsiksi, generoituu käännösvirhe; muussa tapauksessa se käynnistetään sijoituksen toteuttamiseen.
4. Ellei eksplisiittistä ilmentymää ole tehty, suoritetaan jäsenittäinen oletussijoitus.
5. Jäsenittäisessä oletussijoituksessa jokaiseen sisäiseen tai yhdistettyyn tyyppiin sijoitetaan sitä vastaavan jäsenen arvo.
6. Jokaiseen luokkaolion jäseneen käytetään vaiheita 1 — 6 rekursiivisesti, kunnes kaikki

sisäisten ja yhdistettyjen tyyppien tietojäsenet on sijoitettu.

Jos jälleen muokkaamme esimerkiksi Account-luokan määritystä niin, että _name on string-tyyppinen luokkaolion jäsen, kääntäjä toteuttaa jäsenittäisen oletussijoituksen

```
newAcct = oldAcct;
```

aivan kuin se olisi generoinut seuraavan kopioinnin sijoitusoperaattorin:

```
inline Account&
Account::
operator=( const Account &rhs )
{
        _balance = rhs._balance;
        _acct_nmbr = rhs._acct_nmbr;

        // tämä käynnistys on oikein
        // myös ohjelmoijan tasolla.
        // sama lyhennettynä: _name = rhs._name;
        _name.string::operator=( rhs._name );
}
```

Account-luokan jäsenittäinen oletussijoitus on kuitenkin yhä kelpaamaton _acct_nmbr-jäsenen jäsenittäisen kopioinnin takia. Pitää yhä tehdä eksplisiittinen kopioinnin sijoitusoperaattori, mutta nyt sitä pitää uudistaa niin, että se käsittelee _name:a luokkaolion string-jäsenenä:

```
Account&
Account::
operator=( const Account &rhs )
{
        // vartoidaan itseensä sijoittamista
        if ( this != &rhs )
        {
                // käynnistää: string::operator=(const string& )
                _name = rhs._name;
                _balance = rhs._balance;
        }
        return *this;
}
```

Jotta voisimme estää jäsenittäisen kopioinnin kokonaan, teemme saman, kuin minkä teimme estääksemme jäsenittäisen alustuksen: esittelemme operaattorin yksityiseksi emmekä tee operaattorille varsinaista määrittelyä.

Yleensä kopiointimuodostaja ja kopioinnin sijoitusoperaattori tulisi ajatella yksikkönä. Jos vaaditaan toinen niistä, vaaditaan enemmän kuin todennäköisesti niistä myös toinen. Jos kiellämme toisen, pitää meidän todennäköisesti kieltää toinenkin.

Harjoitus 14.17

Tee kopioinnin sijoitusoperaattori jokaiselle luokalle kohdan 14.6 harjoituksessa 14.14.

Harjoitus 14.18

Päättele, tarvitaanko kopioinnin sijoitusoperaattori kohdan 14.2 harjoituksessa 14.3 valitulle luokalle. Jos tarvitaan, tee se. Muussa tapauksessa perustele, miksi se on tarpeeton.

14.8 Tehokkuusnäkökohtia

Yleensä on tehokkaampaa välittää luokkaolio funktiolle joko osoittimen tai viittauksen avulla kuin arvona. Esimerkiksi funktion tunniste

```
bool sufficient_funds( Account acct, double );
```

vaatii, että jokainen käynnistys alustaa jäsenittäin `acct`-parametrin todellisella välitetyllä Account-oliolla. Kumpikin uudistettu funktion tunniste

```
bool sufficient_funds( const Account *pacct, double );
bool sufficient_funds( const Account &acct, double );
```

vaatii yksinkertaisesti, että Account-olion osoite kopioidaan. Mitään luokan alustusta ei tarvitse tapahtua (katso kohdasta 7.3 viittaus- ja osoitinparametrisuhteiden käsittely).

Vaikka on tehokkaampaa palauttaa osoitin tai viittaus luokkaolioon kuin palauttaa luokkaolio arvona, se on huomattavasti vaikeampaa ohjelmoida oikein. Mietitäänpä seuraavaa lisäysoperaattoria:

```
// tekee työn, mutta saattaa olla huomattavan tehoton
// suurilla matrix-olioilla
Matrix
operator+( const Matrix& m1, const Matrix& m2 )
{
        Matrix result;
        // suorita laskenta ...
        return result;
}
```

Ylikuormitettu lisäysoperaattori mahdollistaa, että käyttäjä voi kirjoittaa

```
Matrix a, b;
// ...

// molemmat käynnistävät operator+()-operaattorin
Matrix c = a + b;
a = b + c;
```

Kuitenkin, jos Matrix on suuri ja monimutkainen luokka, result-matriisin palauttaminen arvona voi olla liian kallista — etenkin, jos operaatiota toistetaan usein ja sen havaitaan olevan suorituskyvyn pullonkaula.

Vaikka seuraava uudistettu toteutus parantaa ohjelmamme suorituskykyä havaittavasti

```
// tehokkaampi, mutta osoite on kelvoton return-lauseen jälkeen
//  -- aiheuttaa todennäköisesti suoritusaikaisen virheen
Matrix&
operator+( const Matrix& m1, const Matrix& m2 )
{
        Matrix result;
        // tee lisäys ...
        return result;
}
```

se johtaa myös säännöllisesti suorituksenaikaisiin virheisiin: result:in osoitteen arvo on tuntematon sen funktion päättymisen jälkeen, jossa se on määritelty. (Itse asiassa palautamme viittauksen paikalliseen olioon, joka ei normaalisti ole olemassa funktion päättymisen jälkeen.)

Osoitteen arvo, jonka palautamme, pitää säilyä kelvollisena funktion päättymisen jälkeen. Vaikka seuraava toteutus saa aikaan osoitteen arvon pysymisen

```
// ei tapaa varmistaa, ettei muisti häviä.
// Koska matrix voi olla suuri, häviäminen voi olla merkittävää.
Matrix&
operator+( const Matrix& m1, const Matrix& m2 )
{
        Matrix *result = new Matrix;
        // tee laskenta ...
        return *result;
}
```

se johtaa myös merkittävään muistin häviämiseen: mikään ohjelmanosa ei ole vastuussa delete-lausekkeen käytöstä olioon sen viimeisen käytön jälkeen. Itse asiassa tämä ei ole hyväksyttävää.

Ohjelmoijan ratkaisu on määritellä uudelleen lisäysoperaattori nimettynä funktiona, jonka kolmanteen viittausparametriin tallennetaan tulos:

```
// tämä saa aikaan vaatimamme tehokkuuden,
// mutta ei ole kovin selkeä käyttäjille ...
void
mat_add( Matrix &result,
      const Matrix& m1, const Matrix& m2 )
{
          // laske suoraan result:iin
}
```

Tämä ratkaisee suorituskykyongelman, mutta luokkaa ei voi enää käyttää operaattori syntaksin kanssa eikä olioiden alustamiseen

```
// ei tueta enää
Matrix c = a + b;
```

eikä voi ottaa mukaan alilausekkeisiin:

```
// ei myöskään tueta enää
if ( a + b > c ) ...
```

Kielen kyvyttömyys palauttaa luokkaolioita tehokkaasti katsottiin merkittäväksi heikkoudeksi. Eräs ehdotettu ratkaisu oli kielilaajennus, jossa nimetään palautettava luokkaolio. Esimerkiksi:

```
Matrix
operator+( const Matrix& m1, const Matrix& m2 )
name result
{
        Matrix result;
        //...
        return result;
}
```

Kääntäjä kirjoittaisi funktion sitten uudelleen sisäisesti ja saisi kolmannen viittausparametrin

```
// sisäisesti uudelleenkirjoitettu funktio
// ehdotetussa kielilaajennuksessa
void
operator+( Matrix &result,
      const Matrix& m1, const Matrix& m2 )
{
        // laske suoraan tulos result:iin
}
```

ja muuttaisi jokaisen funktion käytön laskemaan tuloksen suoraan kohdeluokkaolioon. Esimerkiksi

```
Matrix c = a + b;
```

on muunnettuna sisäisesti

```
Matrix c;
operator+( c, a, b );
```

Tästä paluuarvolaajennuksesta ei koskaan tullut osa kieltä — vaan optimoinnista. Tajuttiin, että kääntäjä huomaisi luokkaolion palautuksen ja voisi tehdä paluuarvon muunnoksen ilman eksplisiittistä kielen laajennusta. Esimerkiksi yleisessä funktiomuodossa

```
classType
functionName( paramList )
{
        classType namedResult;
        // tee työ ...
        return namedResult;
}
```

kääntäjä muuntaa sekä funktion että sen käyttötilanteet sisäisesti muotoon

```
void
functionName( classType &namedResult, paramList )
{
          // laske suoraan namedResult:iin
}
```

eliminoiden luokkaolion palauttamisen arvona ja tarpeen käynnistää kopiointimuodostaja. Jotta se voitaisiin laukaista, pitää palautetun luokkaolion olla sama nimetty olio funktion jokaisessa paluupisteessä.

Sitten eräs viimeisistä asioista C++-luokkaolion tehokkuudesta. Luokkaolion alustus kuten

```
Matrix c = a + b;
```

on aina tehokkaampi kuin siihen sijoitus. Esimerkiksi, kun ohjelman tulokset ovat täsmälleen samat ja kirjoitetaan

```
Matrix c;
c = a + b;
```

tämä vaatii huomattavasti enemmän laskentaa saavuttaa nuo tulokset. Samalla tavalla on tehokkaampaa kirjoittaa

```
for ( int ix = 0; ix < size-2; ++ix ) {
      Matrix matSum = mat[ix] + mat[ix+1];
      // ...
}
```

kuin

```
Matrix matSum;
for ( int ix = 0; ix < size-2; ++ix ) {
        matSum = mat[ix] + mat[ix+1];
        // ...
}
```

Se, että sijoitus on aina raskaampaa, johtuu yleensä siitä, että emme voi korvata suoraan palautettavan paikallisen luokkaolion sijoituksen kohdetta. Kun siis

```
Point3d p3 = operator+( p1, p2 );
```

voidaan turvallisesti muuntaa muotoon

```
// C++-pseudokoodia
Point3d p3;
```