Luokat ja funktiot: toteutus

Koska C++-kieli on vahvasti tyypitetty, asianmukaisten määritysten keksiminen luokillesi ja malleillesi ja asianmukaisten määritysten keksiminen funktioillesi muodostavat leijonanosan taistelusta. Kun saat nämä toimimaan oikein, malli-, luokka- ja funktiototeutusten kanssa on vaikea epäonnistua. Ihmiset pystyvät kuitenkin siihenkin.

Samat ongelmat tulevat esiin silloin, kun abstraktiota häväistään huomaamatta: silloin, kun toteutuksen yksityiskohdat satutaan vahingossa päästämään pilkistämään niiden sisältävien luokan ja funktion rajojen takaa. Muut ongelmat ovat peräisin hämmennyksestä, joka koskee olion elinikää. Edelleen, muut juontavat alkunsa ennenaikaisesta optimoinnista, joka on tyypillisesti jäljitettävissä `inline`-avainsanan käytön (avoimet funktiot) viettelevään luonteeseen. Lisäksi eräistä toteutusstrategioista, vaikka ne ovat hyviä paikallisella asteikolla, on tuloksena lähdetiedostojen välinen kaksinkertaistuminen, ja tämä voi tehdä isojen järjestelmien uudelleenrakentamisesta liian kallista.

Kaikki nämä ongelmat, kuten muut vastaavatkin, voidaan välttää, jos tiedetään mitä valvoa. Seuraavissa Kohdissa yksilöidään eräitä tilanteita, joissa sinun kannattaa olla erityisen valppaana.

Kohta 29: Vältä "kahvojen" palauttamista sisäiseen tietoon.

Kohtaus oliopohjaisesta romanssista:

Olio A: Rakkaani, älä koskaan muutu!

Olio B: Älä murehdi, rakkaani. Minä olen `const`-tyyppinen.

Samoin kuin tosielämässä, A ihmettelee, "Voidaanko B-olioon luottaa?" Ja juuri kuin tosielämässä, vastaus on B-olion varassa: sen jäsenfunktioiden rakenne.

Oletetaan, että B on vakio `String` -tyyppinen olio:

```
class String {
public:
  String(const char *value);       // katso Kohdasta 11
  ~String();                       // mahdolliset toteutukset

  operator char *() const;         // muuntaa String -> char*

  ...

private:
  char *data;
};

const String B("Hello World");     // B on const-tyypp. olio
```

Koska B on tyypiltään `const`, B-olion arvon on parasta olla nyt ja ikuisesti "Hello World". Tällä oletetaan tietysti, että B-olion parissa työskentelevät ohjelmoijat työskentelevät sivistyneesti. Se riippuu varsinkin siitä , että kukaan ei "muunna B-olion *const-tyyppisyyttä*" tämänkaltaisten ilkeämielisten syrjähyppyjen avulla (katso Kohta 21):

```
String& alsoB =                  // tee alsoB:stä 2. nimi
  const_cast<String&>(B);        // B:lle, mutta ilman
                                 // const-tyyppisyyttä
```

Kun oletetaan, että kukaan ei tee tällaisia sopimuksia paholaisen kanssa, tuntuu kuitenkin varmemmalta vedolta, että B ei tule koskaan muuttumaan. Vai muuttuuko? Tutki tätä tapahtumasarjaa:

```
char *str = B;                   // kuts. B.operator char*()

strcpy(str, "Hi Mom");           // muuttaa sen mihin str
                                 // osoittaa
```

Onko B-oliolla yhä arvo "Hello World", vai onko se yhtäkkiä muuttunut joksikin, mitä voisit sanoa äidillesi? Vastaus riippuu kokonaan `String::operator char*`-funktion toteutuksesta.

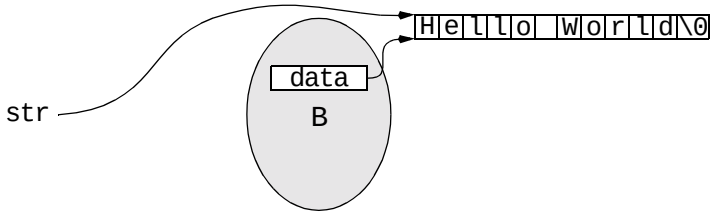
Tässä on huolimaton toteutus, eli se, joka suorittaa asian väärin. Se tekee sen kuitenkin erittäin tehokkaasti, ja tämän takia monet ohjelmoijat putoavat tähän ansaan:

```
// nopea, mutta virheellinen toteutus
inline String::operator char*() const
{ return data; }
```

Tämän funktion rakenteellinen vika on siinä, että se palauttaa "kahvan" - tässä tapauksessa osoittimen - tietoon, jonka pitäisi olla piilotettuna sen `String`-olion sisällä, jonka kohdalla funktiota pyydettiin avuksi. Tuo kahva antaa kutsujilleen rajoittamattoman pääsyn siihen, mihin yksityinen kenttä `data` osoittaa. Toisin sanoen, tämän lauseen jälkeen

```
char *str = B;
```

tilanne näyttää tältä:



Näyttää selvästi siltä, että kaikki `str`-muuttujan osoittamaan muistiin tapahtuvat muutokset muuttavat myös `B`:n tehokasta arvoa. Täten, vaikka `B` on esiteltynä `const`-tyyppisenä, ja vaikka `B`:n kohdalla avuksi pyydetään vain `const`-tyyppisiä jäsenfunktioita, `B` hankkii silti erilaisia arvoja ohjelman suorituksen aikana. Varsinkin silloin, jos `str` muuttaa sitä mihin se osoittaa, myös `B` muuttuu.

Tavassa, jolla `String::operator char*` -funktio on kirjoitettu, ei luonnostaan ole mitään väärää. Siitä voi aiheutua vaikeuksia, että sitä voidaan käyttää vakio-olioihin. Jos funktiota ei olisi esitelty `const`-tyyppisenä, ei olisi mitään ongelmaa, koska sitä ei voitaisi käyttää `B`-olion tyyppisiin olioihin.

`String`-olion, myös vakiotyyppisen, muuttaminen sen vastaavaan `char*`-pohjaiseen muotoon tuntuu kuitenkin täysin järkevältä, joten haluat pitää tämän funktion varmasti `const`-tyyppisenä. Jos haluat tehdä niin, sinun täytyy kirjoittaa toteutuksesi uudelleen välttääksesi kahvan palauttamisen olion sisäiseen tietoon:

```
// hitaampi, mutta turvallisempi toteutus
inline String::operator char*() const
{
  char *copy = new char[strlen(data) + 1];
  strcpy(copy, data);

  return copy;

}
```

Tämä toteutus on turvallinen, koska se palauttaa osoittimen muistiin, joka sisältää *kopion* tiedosta, johon `String`-olio osoittaa; ei ole mitään keinoa muuttaa `String`-olion arvoa tämän funktion palauttamalla osoittimella. Tällaisilla turvallisilla komennoilla on, kuten tavallista, hintansa: tämä versio `String::operator char*`-funktiosta on hitaampi kuin edellä mainittu yksinkertainen versio, ja tämän funktion kutsujien täytyy muistaa käyttää sen palauttamaan osoittimeen `delete`-operaattoria.

Jos sinusta tuntuu, että tämä versio `operator char*` -funktiosta on liian hidas, tai jos mahdollinen muistivuoto saa sinut hermostuneeksi (sen pitäisikin), lievästi erilainen tapa on palauttaa osoitin `char`-tyyppisiin *vakioihin*:

```
class String {
public:
   operator const char *() const;

  ...

};

inline String::operator const char*() const
{ return data; }
```

Tämä funktio on nopea ja turvallinen, ja vaikka se ei ole sama kuin se, jonka alunperin määrittelit, se riittää useimmille sovelluksille. Se on myös moraalinen vastine C++-kielen standardointikomitean ratkaisulle `string/char*`-arvoitukseen: standardi `string`-tyyppi sisältää `c_str`-jäsenfunktion, joka palauttaa `const char *` -version kyseessä olevasta `string`-muuttujasta. Jos haluat lisätietoja perus-`string`-tyypistä, katso Kohta 49.

Osoitin ei ole ainoa tapa palauttaa kahva sisäiseen tietoon. Viittauksia on aivan yhtä helppo käyttää väärin. Tässä on yleinen tapa tehdä se, käyttämällä jälleen `String`-luokkaa:

```
class String {
public:

  ...

  char& operator[](int index) const
 { return data[index]; }

private:
  char *data;
};

String s = "I'm not constant";

s[0] = 'x';                    // toimii, s ei ole const

const String cs = "I'm constant";

cs[0] = 'x';                   // tämä muuttaa const-tyyp.
                               // merkkijonon, mutta
                               // kääntäjät eivät huomaa
```

Huomaa, kuinka `String::operator[]`-funktio palauttaa tuloksensa viittausparametrillä. Tämä tarkoittaa sitä, että tämän funktion kutsuja saa takaisin sisäisen elementin `data[index]` *toisen nimen*, ja tätä toista nimeä voidaan käyttää muuttamaan olettavasti vakio-olion sisäinen tieto. Tämä on sama ongelma kuin mihin törmäsit aikaisemmin, mutta tällä kertaa syytetty on viittaus palautuksen arvona, ei osoittimena.

Tämänkaltaisen ongelman yleiset ratkaisut ovat samat kuin mitä ne ovat osoittimille: tee funktiosta joko ei-`const`-tyyppinen, tai kirjoita se uudelleen niin, että mitään kahvaa ei palauteta. Ratkaisu juuri tähän *tiettyyn* ongelmaan - kuinka kirjoittaa `String::operator[]`-funktio niin, että se toimii sekä `const`- että ei-`const`-olioille - katso Kohta 21.

`const`-tyyppiset jäsenfunktiot eivät ole ainoita, joiden täytyy huolehtia kahvojen palauttamisesta. Myös ei-`const`-jäsenfunktioiden täytyy sopeutua siihen tosiasiaan, että kahvan kelpoisuus loppuu samalla hetkellä kuin sitä vastaavan olion. Tämä voi tapahtua nopeammin kuin asiakas odottaa, varsinkin silloin, kun kyseessä oleva olio on kääntäjän luoma tilapäinen olio.

Tutki esimerkiksi tätä funktiota, joka palauttaa `String`-olion.

```
String someFamousAuthor()          // satunnaisesti valitsee
{                                  // ja pal. kirjoit. nimen
  switch (rand() % 3) {            // rand() on <stdlib.h>
                                   // (ja <cstdlib> – katso
                                   // Kohta 49)
  case 0:
    return "Margaret Mitchell";    // Kirj. "Tuulen Viemää",
                                   // todellisen klassikon
  case 1:
    return "Stephen King";         // Hänen tarinansa ovat
                                   // pitäneet miljoonia
                                   // unettomina öisin
  case 2:
    return "Scott Meyers";         // Oho, yksi näistä
  }                                // jutuista ei kuulu
                                   // muiden joukkoon...
  return "";                       // emme pääse tänne, mutta
                                   // kaikki arvonpalautta-
                                   // vassa funktiossa
}                                  // olevat polut palautta-
                                   // vat arvon, huokaus
```

Laita kiltisti sivummalle huolesi siitä, kuinka "satunnaisia" `rand`-funktion palauttamat luvut ovat, ja suhtaudu huumorilla suurpiirteisyyden harhoihini, joissa yhdistän itseni todellisiin kirjailijoihin. Keskity sen sijaan siihen tosiasiaan, että `someFamousAuthor`-funktion paluuarvo on `String`-olio. Tällaiset oliot ovat hetkellisiä - niiden elinkaari pitenee yleensä vain sen ilmaisun loppuun, joka sisältää kutsun funktioon, joka luo ne. Tämä olisi tässä tapauksessa sen ilmaisun loppuun, joka sisältää kutsun `someFamousAuthor`-funktioon.

Tutki seuraavaksi tätä esimerkkiä `someFamousAuthor`-funktion käytöstä, jossa oletamme, että `String`-luokka esittelee `operator const char*` -jäsenfunktion alla kuvatulla tavalla:

```
const char *pc = someFamousAuthor();
cout << pc;                          // oh oh...
```

Usko tai älä, et voi ennustaa, mitä tämä koodi tekee, et ainakaan millään varmuudella. Tämä johtuu siitä, että sillä hetkellä, kun yrität tulostaa `pc`:l-osoittimelle osoitetun merkkijonojakson, tuo jakso on määrittelemätön. Vaikeudet saavat alkunsa tapahtumista, jotka tulevat ilmi `pc`-osoittimen alustuksen aikana:

1. Tilapäinen `String`-olio luodaan tallentamaan `someFamousAuthor`-funktion paluuarvo.
2. Tuo `String`-olio muunnetaan `const char*` -tyyppiseksi arvoksi `String`-olion `operator const char*` -jäsenfunktion kautta, ja `pc` alustetaan tuloksena olevalla osoittimella.
3. Tilapäinen `String`-olio tuhotaan, mikä tarkoittaa sitä, että sen tuhoajafunktiota kutsutaan. Sen `data`-osoitin poistetaan tuon tuhoajafunktion sisällä (koodi nähdään Kohdassa 11). `data` osoittaa kuitenkin samaan muistiin kuin `pc`, joten `pc` osoittaa nyt poistettuun muistiin - muistiin, jossa on määrittelemätön sisältö.

Koska `pc` alustettiin tilapäiseen olioon kohdistuvalla kahvalla ja tilapäiset oliot poistetaan pian niiden luonnin jälkeen, kahvasta tuli viallinen ennen kuin `pc` pystyi tekemään sillä mitään. Kaikkine aikeineen ja tarkoituksineen `pc` oli kuollut jo syntyessään. Tällainen on tilapäisissä olioissa käytettävien kahvojen vaara.

Kahvojen palauttaminen `const`-tyyppisille jäsenfunktioille on väärää neuvontaa, koska se vahingoittaa abstraktiota. Kahvojen palauttaminen jopa ei-`const`-tyyppisille jäsenfunktioille voi kuitenkin johtaa vaikeuksiin, varsinkin silloin, kun tilapäiset oliot sotkeutuvat mukaan. Kahvat voivat "roikkua" kuten osoittimetkin, ja samalla tavalla kuin työskentelet välttääksesi roikkuvia osoittimia, sinun kannattaa myös pyrkiä välttämään roikkuvia kahvoja.

Ei silti ole syytä heittäytyä fasistiseksi tästä. Kaikkia mahdollisia roikkuvia osoittimia ei ole mahdollista tömistää esiin vähäpätöisissä ohjelmissa, ja kaikkien mahdollisten roikkuvien kahvojen eliminoiminen on myös harvoin mahdollista. Joka tapauksessa, jos yrität välttää kahvojen palauttamista silloin, kun ei ole mitään pakottavaa syytä, ohjelmasi hyötyvät siitä ja samoin maineesi.

Kohta 30: Vältä jäsenfunktioita, jotka palauttavat ei-`const`-osoittimia tai viittauksia jäseniin, jotka ovat huonommin saavutettavissa kuin ne itse.

Syy siihen, että jäsenestä tehdään yksityinen tai suojattu, on rajoittaa pääsy siihen, eikö niin? Sinun ylityöllistetyt ja alipalkatut C++-kääntäjäsi joutuvat moniin vaikeuksiin varmistaessaan, että pääsyrajoituksiasi ei ole juonittu, eikö niin? Ei ole siis järkevää kirjoittaa funktioita, jotka antavat satunnaisille asiakkaille nyt kyvyn päästä vapaasti rajoitettuihin jäseniin, eihän? Jos se sinusta tuntuu järkevältä, lue tämä kappale uudelleen ja uudelleen, kunnes olet samaa mieltä, että siinä ei ole järkeä.

Tätä yksinkertaista sääntöä on helppo vahingoittaa. Tässä on esimerkki:

```
class Address { ... };          // jossa joku asuu

class Person {
public:
  Address& personAddress() { return address; }
  ...

private:
  Address address;
  ...
};
```

Jäsenfunktio `personAddress` tarjoaa kutsujalle `Address`-olion, joka sijaitsee `Person`-oliossa, mutta mahdollisesti tehokkuusharkinnan takia tulos palautetaan arvoparametrin sijasta viittausparametrinä (katso Kohta 22). Valitettavasti tämän jäsenfunktion olemassaolo mitätöi `Person::address`-funktion tekemisen yksityiseksi:

```
Person scott(...);              // parametrit jätetty pois
                                // yksinkertaisuuden takia

Address& addr =                 // oletetaan, että addr on
  scott.personAddress();        // globaalinen
```

Globaali olio `addr` on nyt *toinen nimi* `scott.address`-oliolle, ja sitä voidaan tarvittaessa käyttää lukemaan ja kirjoittamaan `scott.address`. `scott.address` ei ole enää `private`-tyyppinen käytännön tarkoituksiin; se on julkinen, ja tämän arvonylennyksen lähde saatavuuteen on jäsenfunktio `personAddress`. `private` ei tietenkään ole mitään erikoista tämän esimerkin saatavuustasossa; jos `address` olisi suojattu, tarkalleen sama järkeily pätisi.

Viittaukset eivät ole ainoa huolenaihe. Osoittimet osaavat myös pelata tätä peliä. Tässä on sama esimerkki, mutta siinä käytetään tällä kertaa osoittimia:

```
class Person {
public:
  Address * personAddress() { return &address; }
  ...
private:
  Address address;
  ...
};
Address *addrPtr =
  scott.personAddress();       // sama ongelma kuin edellä
```

Kun puhutaan osoittimista, sinun pitää kuitenkin huolehtia tietojäsenien lisäksi jäsen-*funktioista*. Tämä siitä syystä, että jäsenfunktioon voidaan palauttaa osoitin:

```
class Person;                  // esittely
// PPMF = PPMF on lyh. "pointer to Person member function"
typedef void (Person::*PPMF)();
class Person {
public:
  static PPMF verificationFunction()
  { return &Person::verifyAddress; }
  ...
private:
  Address address;
  void verifyAddress();
};
```

Jos et ole tottunut sosiaaliseen kanssakäymiseen jäsenfunktioiden ja niistä johtuvien typedef-komentojen osoittimien kanssa, `Person::verificationFunction`-funktion esittely voi tuntua kesyltä. Älä pelkää. Tässä sanotaan vain, että

- `verificationFunction` on jäsenfunktio, joka ei ota vastaan parametrejä;
- sen paluuarvo on osoitin `Person`-luokan jäsenfunktioon;
- osoitettu funktio (toisin sanoen `verificationFunction`-funktion paluuarvo) ei ota parametrejä eikä palauta mitään, eli `void`-tyyppiä.

Mitä tulee `static`-sanaan, se tarkoittaa samaa kuin aina jäsenen esittelyssä: koko luokalla on vain yksi kopio jäsenestä, ja jäseneen päästään käsiksi ilman oliota. Jos haluat lukea koko tarinan, tutki omaa suosikkiasi C++-kielen esittelykirjoista. (Jos omassa suosikki-C++-kirjassasi ei käsitellä staattisia jäseniä, revi varovasti irti kaikki sivut ja pane ne kierrätykseen. Poista kirjan kansi ympäristöystävällisesti jämerällä tavalla, ja lainaa tai osta sitten parempi tekstikirja.)

`verifyAddress` on tässä viimeisessä esimerkissä yksityinen jäsenfunktio, joka osoittaa, että se on itse asiassa luokan toteutuksen yksityiskohta; vain luokan jäsenien pitäisi tietää siitä (ja ystäväfunktioiden tietenkin myös). Julkinen jäsenfunktio `verificationFunction` palauttaa kuitenkin osoittimen `verifyAddress`-funktioon, joten asiakkaat voivat jälleen kiskaista tämänkaltaisen jutun:

```
PPMF pmf = scott.verificationFunction();
(scott.*pmf)();                 // sama jos kutsutaan
                                // scott.verifyAddress
```

`pmf`-funktiosta on tässä tullut synonyymi `person::verifyAddress`-funktiolle, sillä ratkaisevalla erolla, että sen käytössä ei ole mitään rajoituksia.

Kohtaat edellä käydystä keskustelusta huolimatta jonakin päivänä tilanteen, jossa, painostettuna saamaan aikaan suorituksen rajoitukset, sinun täytyy rehellisesti kirjoittaa jäsenfunktio, joka palauttaa viittausparametrin tai osoittimen huonommin saavutettavissa olevaan jäseneen. Sinä et kuitenkaan samalla halua uhrata pääsyn rajoituksia, jotka `private` ja `protected` tarjoavat sinulle. Näissä tapauksissa molemmat tavoitteet voidaan melkein tavoittaa palauttamalla osoitin tai viittausparametri `const`-tyyppiseen olioon. Jos haluat yksityiskohtia, katso Kohta 21.

Kohta 31: Älä koskaan palauta viittausparametriä paikalliseen olioon tai viittaamattomaan osoittimeen, joka on funktion sisäpuolella alustettu `new`-operaattorilla.

Tämä Kohta voi tuntua monimutkaiselta, mutta se ei ole sitä. Tämä on yksinkertaisesti yleistä tietoa. Todella. Varmasti. *Luota minuun*.

Tutki ensin tilannetta, jossa viittaus palautetaan paikalliseen olioon. Ongelma on juuri siinä, että paikalliset oliot ovat juuri niitä, *paikallisia*. Tämä tarkoittaa sitä, että ne muodostetaan silloin, kun ne määritetään, ja ne tuhotaan silloin, kun ne menevät näkyvyysalueensa ulkopuolelle. Niiden näkyvyysalue on kuitenkin sen funktion runko, jossa ne sijaitsevat. Kun funktio palauttaa, hallinta poistuu näkyvyysalueelta, joten oliot, jotka ovat paikallisia tuolle funktiolle, tuhotaan automaattisesti. Tästä on tuloksena, että jos palautat viittauksen paikalliseen olioon, tuo paikallinen olio on tuhottu ennen kuin funktion kutsuja pääsee laskennallisilla käsillään sen kimppuun.

Tämä ongelma tulee ilkeästi ilmi usein silloin, kun yrität kehittää funktiosi tehokkuutta palauttamalla sen tuloksen viittausparametrillä arvoparametrin sijasta. Seuraava esimerkki on sama kuin Kohdassa 23 oleva. Se esittää yksityiskohtaisen kysymyksen siitä, milloin voit palauttaa viittauksen ja milloin et:

```
class Rational {         //suhdelukujen luokka
public:
  Rational(int numerator = 0, int denominator = 1);
  ~Rational();

  ...

private:
  int n, d;              // osoittaja ja nimittäjä

// huomaa, että operator* palauttaa (väärin) viittauksen
friend const Rational& operator*(const Rational& lhs,
                                 const Rational& rhs);
};

// operator*-funktion väärä toteutus
inline const Rational& operator*(const Rational& lhs,
                                 const Rational& rhs)
{
  Rational result(lhs.n * rhs.n, lhs.d * rhs.d);
  return result;
}
```

Paikallinen `result`-olio on tässä muodostettu silloin, kun se kirjoitetaan `operator*`-funktion runkoon. Paikalliset oliot tuhotaan kuitenkin automaattisesti silloin, kun ne menevät näkyvyysalueensa ulkopuolelle. `result` menee näkyvyysalueensa ulkopuolelle `return`-lauseen suorituksen jälkeen, joten kun kirjoitat tämän,

```
Rational two = 2;

Rational four = two * two;     // sama kuin
                               // operator*(two, two)
```

funktio-kutsun aikana tapahtuu tämä:

1. Paikallinen olio `result` muodostetaan.
2. Viittaus alustetaan toisena nimenä `result`-oliolle ja tämä viittaus hupsutellaan pois `operator*`-funktion paluuarvona.
3. Paikallinen `result`-olio tuhotaan, ja sen valtaama tila pinossa tehdään ohjelman tai muiden ohjelmien muiden osien käytettäväksi.
4. Olio `four` alustetaan käyttämällä vaiheen 2 viittausta.

Kaikki menee hienosti ennen vaihetta 4, jolloin tapahtuu, kuten hienoimmissa korkeatekniikan piireissä sanotaan, "pääasiallinen menetys". Vaiheessa 2 alustettu viittaus lakkasi viittaamasta oikeaan olioon kuten vaiheen 3 lopussa, joten olion `four` alustuksen tulos on täysin määrittelemätön.

Opetuksen pitäisi olla selvä: älä palauta viittausta paikalliseen olioon.

"Selvä", sanot, "ongelma on vain siinä, että olio, jota haluan käyttää, menee näkyvyysalueensa ulkopuolelle liian pian. Voin korjata asian. Kutsun `new`-funktiota sen sijaan, että käyttäisin paikallista oliota." Tähän tyyliin:

```
// toinen väärä operator*-funktion toteutus
inline const Rational& operator*(const Rational& lhs,
                                 const Rational& rhs)
{
  // luo uusi olio kekoon
  Rational *result =
    new Rational(lhs.n * rhs.n, lhs.d * rhs.d);

  // palauta se
  return *result;
}
```

Tämä työtapa todellakin välttää edellisen esimerkin ongelman, mutta tilalle tulee toinen ongelma. Kun haluat välttää muistivuodon ohjelmassasi, tiedät, että sinun täytyy varmistaa, että `delete`-operaattoria sovelletaan jokaiseen osoittimeen, jonka `new` on loihtinut esiin, mutta ikävä kyllä on kuitenkin hankaluus: kuka suorittaa vastaavan `delete`-kutsun, kun tätä funktiota käytetään `new`-operaattoriin?

Funktion `operator*` *kutsujan* täytyy selvästi varmistaa se, että `delete`-operaattoria käytetään. Selvää kyllä, ja jopa helppoa dokumentoida, mutta joka tapauksessa toivotonta. Tähän pessimistiseen arvioon on kaksi syytä.

Ensinnäkin, on hyvin tunnettua, että ohjelmoijat ovat lajina hutiloivia. Tämä ei tarkoita sitä, että sinä tai minä olen hutiloiva, mutta harvassa ovat ne ohjelmoijat, jotka eivät työskentele sellaisten kanssa, jotka ovat - voinko sanoa? - hieman epäsovinnaisia. Mitkä ovat mahdollisuudet, että tällaiset ohjelmoijat - ja tiedämme kaikki, että heitä on - muistavat, että aina kun he kutsuvat `operator*`-funktiota, heidän *täytyy ottaa tuloksen osoite* ja käyttää sitten `delete`-operaattoria siihen? Tämä tarkoittaa sitä, että heidän täytyy käyttää `operator*`-funktiota tähän tyyliin:

```
const Rational& four = two * two;   // ota osoittamaton
                                    // osoitin; tallenna se
                                    // viittaukseen
...
delete &four;                       // nouda osoitin
                                    // ja poista se
```

Mahdollisuudet ovat häviävän pienet. Muista, että jos vain *yksi* `operator*`-funktion *kutsuja* epäonnistuu sääntöjen noudattamisessa, tuloksena on muistivuoto.

Kun viittaamattomia osoittimia palautetaan, on toinen, vakavampi ongelma, joka ei hellitä, vaikka paikalla olisi kaikkein omantunnontarkimmat ohjelmoijat. Funktion `operator*` tuloksena on usein tilapäinen väliarvo, olio, joka on olemassa vain niihin tarkoituksiin, joissa arvotetaan laajempi ilmaisu. Esimerkiksi:

```
Rational one(1), two(2), three(3), four(4);
Rational product;

product = one * two * three * four;
```

Sen ilmaisun arvottaminen, joka sijoitetaan `product`-luokkaan, vaatii kolme erillistä kutsua `operator*`-funktioon. Tämä on tosiasia, joka tulee vielä ilmeisemmäksi, kun kirjoitat ilmaisun uudelleen vastaavassa toiminnallisessa muodossaan:

```
product = operator*(operator*(operator*(one, two), three), four);
```

Tiedät, että jokainen `operator*`-funktion kutsuista palauttaa olion, joka pitää poistaa, mutta `delete`-operaattoria ei ole mahdollista ottaa käyttöön, koska palautetuista olioista ei ole yhtäkään tallennettu minnekään.

Tämä hankaluus voidaan ratkaista vain pyytämällä asiakkailta tämänkaltaista lähdekoodia:

```
const Rational& temp1 = one * two;
const Rational& temp2 = temp1 * three;
const Rational& temp3 = temp2 * four;

delete &temp1;
delete &temp2;
delete &temp3;
```

Tee näin ja paras, mitä voit toivoa on, että ihmiset ovat välittämättä sinusta. Realistisempaa on, että sinut tullaan nylkemään elävänä, tai mahdollisesti passittamaan kymmenen vuoden pakkotyöhön kirjoittamaan mikrokoodia vohveliraudoille ja leivänpaahtimille.

Ota opiksi nyt ja sen jälkeen: funktion kirjoittaminen, joka palauttaa viittaamattoman osoittimen, on sama kuin muistivuoto, joka vain odottaa tapahtumistaan.

Muuten, jos luulet, että olet keksinyt tavan, jolla voidaan välttää määrittelemätön käytös, joka periytyy siitä, että palautetaan viittaus paikalliseen olioon ja muistivuoto, joka kummittelee paluuarvoa viittauksesta keossa varattuun olioon, palaa Kohtaan 23 ja lue, miksi viittauksen palauttaminen paikalliseen `static`-tyyppiseen olioon ei myöskään tule toimimaan kunnolla. Se voi säästää sinua vaivalta, kun etsit lääkettä käsivarrellesi, joka rasittuu, kun yrität taputella itseäsi selkään.

Kohta 32: Siirrä muuttujien määrityksiä niin pitkälle kuin mahdollista.

Olet siis tilannut itsellesi sen C-filosofian, että muuttujat pitäisi määrittää lohkon alkupäässä. Peruuta tuo tilaus! C++-kielessä se on tarpeeton, luonnoton ja kallis.

Muista, että kun määrität muuttujan sillä tyypillä, jolla on muodostinfunktio tai tuhoajafunktio, muodostamisen kustannukset koituvat sinulle velaksi silloin, kun hallinta saavuttaa muuttujan määrityksen, ja hajottamisen kustannukset silloin, kun muuttuja menee näkyvyysalueensa ulkopuolelle. Tämä tarkoittaa sitä, että käyttämättömiin muuttujiin sisältyy kustannus, joten haluat varmaan välttää niitä aina kun voit.

Tiedän sinun olevan joustava ja hienostunut ohjelmointitavoissasi, ja ajattelet varmaan, että et koskaan määritä käyttämättömiä muuttujia, joten tämän Kohdan ohje on sovellettavissa tiukkaan ja niukkaan koodaustyyliisi. Sinun kannattaa kuitenkin miettiä uudelleen. Tutki seuraavaa funktiota, joka palauttaa kryptatun version salasanasta, olettaen että salasana on tarpeeksi pitkä. Jos salasana on liian lyhyt, funktio muodostaa poikkeuksen, joka on tyyppiä `logic_error`, tämä on määritetty normaalissa C++-kirjastossa (katso Kohta 49):

```
// tämä funktio määrittelee muutt. "encrypted" liian pian
string encryptPassword(const string& password)
{
  string encrypted;

  if (password.length() < MINIMUM_PASSWORD_LENGTH) {
     throw logic_error("Salasana on liian lyhyt");
  }

  tee se, mikä on välttämätöntä encrypted-version
  password-muuttujasta sijoittamis. encrypted-muuttujaan;

  return encrypted;
}
```

`encrypted`-olio ei ole *täysin* käyttämätön tässä funktiossa, mutta se on käyttämätön, jos poikkeus muodostetaan. Tämä tarkoittaa sitä, että maksat `encrypted`-olion muodostamisen ja tuhoamisen kustannukset vaikka `encryptPassword`-muuttuja muodostaisi poikkeuksen. Tästä on tuloksena, että sinun kannattaa siirtää `encrypted`-muuttujan määritystä, kunnes tiedät, että tarvitset sitä:

```
// tämä funktio siirtää "encrypted"-muuttujan määritystä
// kunnes se on todella välttämätöntä
string encryptPassword(const string& password)
```

```
{
  if (password.length() < MINIMUM_PASSWORD_LENGTH) {
    throw logic_error("Password is too short");
  }

  string encrypted;

  tee se, mikä on välttämätöntä encrypted-version
  vpassword-muuttujasta sijoittamis. encrypted-muuttujaan;

  return encrypted;
}
```

Tämä koodi ei silti ole niin tiukkaa kuin se voisi olla, koska `encrypted`-muuttuja on määritetty ilman alustusargumentteja. Tämä tarkoittaa sitä, että oletuksena olevaa muodostinfunktiota käytetään. Monissa tapauksissa annat oliolle ensimmäiseksi jonkin arvon, usein sijoituksen kautta. Kohdassa 12 selitetään, miksi olion muodostaminen oletuksen mukaisesti ja sen sijoittaminen sitten on paljon tehottomampaa kuin sen alustaminen juuri haluamallasi arvolla. Tämä analyysi pätee myös tässä. Oletetaan esimerkiksi, että `encryptPassword`-funktion vaikea osa suoritetaan tässä funktiossa:

```
void encrypt(string& s);       // salaa s paikallaan
```

`encryptPassword`-funktio voitaisiin toteuttaa tällä lailla, vaikka se ei olisikaan paras tapa tehdä se:

```
// tämä funktio lykkää "encrypted"-muuttujan määritystä
// kunnes se on todella välttämätöntä, mutta se on silti
// tarpeettoman tehoton
string encryptPassword(const string& password)
{
  ...                          // tark. pit. kuten edell.

  string encrypted;            // oletusmuod. encrypted
  encrypted = password;        // sijoita encrypted-muutt.
  encrypt(encrypted);
  return encrypted;
}
```

Suositeltavampi työtapa on alustaa `encrypted password`-muuttujalla, ohittaen täten (tarkoituksettoman) oletusmuodostuksen:

```
// lopuksi, paras tapa määrittää ja alustaa encrypt.-muutt.
string encryptPassword(const string& password)
{
  ...                          // tarkista pituus

  string encrypted(password);  // määritä ja alusta
                               // kopiomuodostimen kautta

  encrypt(encrypted);
  return encrypted;
}
```

Tämä viittaa tämän Kohdan otsikon lauseen "niin kauas kuin mahdollista" todelliseen merkitykseen. Sinun ei pitäisi pelkästään siirtää muuttujan määrittämistä oikeaan hetkeen ennen kuin sinun pitää käyttää muuttujaa, sinun pitäisi myös yrittää siirtää määrittämistä kunnes sinulla on muuttujan alustuksen argumentit. Kun teet näin, et pelkästään vältä tarpeettomien olioiden muodostamista ja tuhoamista, vaan vältät myös tarkoituksettomat oletusmuodostukset. Helpotat lisäksi muuttujien käytön dokumentointia alustamalla ne siinä yhteydessä, missä niiden tarkoitus on selkeä. Muistatko, että C-kielessä sinua rohkaistaan sijoittamaan lyhyt kommentti jokaisen muuttujamäärityksen jälkeen selvittämään sitä, mihin muuttujaa lopulta käytetään? Yhdistä kunnolliset muuttujanimet (katso myös Kohta 28) asiayhteydeltään mielekkäisiin alustusargumentteihin, ja sinulla on jokaisen ohjelmoijan unelma: vankka argumentti eräiden kommenttien *eliminoimiseen*.

Kun siirrät muuttujien määritystä, kehität ohjelmasi tehokkuutta, lisäät ohjelmasi selkeyttä ja vähennät tarvetta dokumentoida muuttujien merkitystä. Näyttää siltä, että on aika jättää jäähyväiset näille lohkon avaaville muuttujamäärityksille.

Kohta 33: Käytä avointen funktioiden muodostamista arvostelukykyisesti.

Avoimet funktiot - mikä *loistava* ajatus! Ne näyttävät funktioilta, ne toimivat kuin funktiot, ne ovat usein paljon parempia kuin makrot (katso Kohta 1), ja voit kutsua niitä ilman, että joudut maksamaan funktiokutsun kustannukset. Voisitko mahdollisesti enää pyytää enempää?

Itse asiassa saat enemmän kuin luulisit, koska funktiokutsun kustannusten välttäminen on vain puolet tarinasta. Kääntäjän optimointirutiinit on tyypillisesti suunniteltu keskittymään sen lähdetekstin ponnistuksiin, joista puuttuvat funktiokutsut, joten kun teet funktiosta avoimen, sallit ehkä kääntäjien suorittaa asiayhteyteen liittyviä optimointeja funktion runko-osassa. Tällaiset optimoinnit eivät olisi mahdollisia "tavallisille" funktiokutsuille.

Ei kannata kuitenkaan innostua liikaa. Ohjelmoinnissa, kuten tosielämässäkään, ei ole ilmaista lounasta, ja avoimet funktiot eivät ole poikkeus. Avointen funktioiden pääajatus on korvata funktion jokainen kutsu lähdetekstin runko-osalla. Et tarvitse tilastotieteen tohtorin arvosanaa nähdäksesi, että tämä tulee todennäköisesti kasvattamaan oliosi lähdetekstin kokoa. Koneissa, joissa on rajoitetusti muistia, liian innokas avointen funktioiden muodostaminen voi aiheuttaa jotakin ohjelmille, jotka ovat liian isoja saatavissa olevaan tilaan. Vaikka saatavilla olisi virtuaalimuistia, avointen funktioiden käytön pakottama lähdetekstin pöhötys voi johtaa patologisesti sivutuskäyttäytymiseen (tuhoamiseen), joka hidastaa ohjelmasi mateluksi. (Se tarjoaa kuitenkin levyohjaimellesi mukavan harjoituskuurin.) Liika avoimuus voi myös vähentää käskysi välimuistin osumatarkkuutta, vähentäen täten käskyn haun nopeutta välimuistista pääasialliseen muistiin.

Jos avoimen funktion runko on toisaalta *erittäin* lyhyt, funktion rungolle luotu koodi voi itse asiassa olla pienempi kuin koodi, joka on kehitetty funktiokutsulle. Jos näin on, funktion määrittäminen avoimeksi voi johtaa *pienempään* oliokoodiin ja parempaan osumatarkkuuteen välimuistissa!

Kannattaa muistaa, että `inline`-direktiivi, kuten `register`, on *vihje* kääntäjille, ei käsky. Tämä tarkoittaa sitä, että kääntäjät ovat vapaita olemaan välittämättä avointen funktioiden direktiiveistä aina kun haluavat, eikä ole kovin vaikeaa saada niitä haluamaan olemaan välittämättä. Useimmat kääntäjät kieltäytyvät esimerkiksi käsittelemästä avoimina "monimutkaisia" funktioita (toisin sanoen niitä, jotka sisältävät silmukoita tai ovat rekursiivisia), ja kaikki paitsi kaikkein turhimmat virtuaalifunktioiden kutsut pysäyttävät avoimuuden rutiinit lähtökuoppiinsa. (Tämän ei pitäisi olla yllätys. `virtual` tarkoittaa, että "odota suoritusta, niin näet mitä funktiota kutsutaan", ja `inline` tarkoittaa, että "korvaa kutsupaikka käännöksen aikana kutsutulla funktiolla". Jos kääntäjät eivät tiedä, mitä funktiota tullaan kutsumaan, voit tuskin syyttää niitä, jos ne kieltäytyvät tekemästä avointa kutsua niihin.) Yhteenveto tästä kaikesta: se, onko annettu avoin funktio todella avoin, on riippuvainen käyttämäsi kääntäjän toteutustavasta. Onneksi useimmilla kääntäjillä on diagnostinen taso, josta on tuloksena varoitus (katso Kohta 48), jos ne epäonnistuvat tekemään avoimia niistä funktioista, joita olet pyytänyt.

Oletetaan, että olet kirjoittanut jonkun funktion `f` ja esitellyt sen avoimena funktiona. Mitä tapahtuu, jos kääntäjä jostain syystä päättää, että se ei tee funktiosta avointa? Ilmeinen vastaus on, että `f` tullaan käsittelemään suljettuna funktiona: `f`:n lähdeteksti luodaan, niin kuin se olisi normaali "suljettu" funktio, ja `f`-funktioon kohdistuvat kutsut käsitellään samalla tavalla kuin normaalit funktiokutsut.

Tämä on teoriassa juuri sitä, mitä tulee tapahtumaan, mutta tämä on yksi niistä tilanteista, jolloin teoria ja käytäntö voivat mennä eri suuntiin. Tämä johtuu siitä, että tämä erittäin siisti ratkaisu ongelmaan, mitä tehdään "suljetuille avoimille funktioille", lisättiin C++-kieleen suhteellisen myöhään standardointiprosessissa. Kielen aikaisemmat määrittelyt (kuten ARM - katso Kohta 50) neuvoivat kääntäjien myyjiä toteuttamaan erilaisen käyttäytymisen, ja vanhempi käyttäytyminen on yhä niin yleistä, että sinun täytyy ymmärtää mikä se on.

Mieti tätä minuutti ja ymmärrät, että avoimet funktiomääritykset sijoitetaan käytännöllisesti katsoen aina otsikkotiedostoihin. Tämä sallii useiden käännösyksiköiden (lähdetiedostot) sisältää samat otsikkotiedostot ja niittää niihin määritettyjen avointen funktioiden hyöty. Tässä on esimerkki, jossa noudatin sitä sopimusta, että lähdetekstitiedostot loppuvat tarkentimeen ".cpp"; tämä on C++-kielen maailmassa ehkä vallitsevin tiedostojen nimeämiskäytäntö:

```
// Tämä on tiedosto example.h
inline void f() { ... }        // f:n määritys

...


// Tämä on tiedosto source1.cpp
#include "example.h"           // sisältää f:n määrityksen

...                            // sisältää kutsun f-funkt.


// Tämä on tiedosto source2.cpp
#include "example.h"           // sis. myös f:n määrityks.
...                            // kutsuu myös f-funktiota
```

Kun sääntönä on vanhat "suljetut avoimet funktiot" ja oletuksena on, että `f` *ei* ole avoin, `source1.cpp` käännetään, ja tuloksena oleva oliotiedosto sisältää funktion nimeltä `f`, juuri samalla tavalla kuin `f`-funktiota ei olisi koskaan esitelty avoimena funktiona. Vastaavasti, kun `source2.cpp` käännetään, sen luoma oliotiedosto sisältää myös funktion nimeltä `f`. Kun yrität linkittää nämä kaksi oliotiedostoa keskenään, voit kohtuudella odottaa linkkerisi valittavan, että ohjelmasi sisältää kaksi määritystä funktioon `f` eli seurauksena on virhe.

Jotta estettäisiin tämä ongelma, vanhat säännöt asettavat, että kääntäjät kohtelevat suljetuksi määritettyä avointa funktiota samalla tavalla, kuin funktio olisi esitelty `static`-tyyppisenä - tämä tarkoittaa paikallista tiedostolle, joka on sillä hetkellä kääntämisen kohteena. Kääntäjät, jotka noudattavat vanhoja sääntöjä, kohtelivat juuri näkemässäsi esimerkissä `f`-funktiota niin kuin se olisi staattinen `source1.cpp`-tiedostossa silloin, kun tämä käännettiin, ja `source2.cpp`-tiedostossa silloin, kun tämä käännettiin. Tämä strategia eliminoi linkittämisen ajan ongelman, mutta sillä on hintansa: jokainen käännösyksikkö, joka sisältää `f`-funktion määrityksen (ja joka kutsuu `f`-funktiota), sisältää *oman* staattisen kopion `f`-funktiosta. Jos `f` itse määrittelee paikallisia staattisia muuttujia, jokainen `f`-funktion kopio saa oman kopion muuttujista. Tämä tulee taatusti hämmästyttämään ohjelmoijia, jotka luulevat, että "`static`" tarkoittaa funktion sisällä samaa kuin "vain yksi kopio".

Tämä johtaa tyrmäävään oivallukseen. Sekä uusien että vanhojen sääntöjen vallitessa, jos avoimesta funktiosta ei ole tehty avointa, sinun täytyy *silti* maksaa jokaisen funktiokutsun kustannukset jokaisesta kutsupaikasta. Vanhojen sääntöjen vallitessa kärsit *myös* lähdetekstin koon kasvamisesta, koska jokainen käännösyksikkö, joka sisältää ja kutsuu `f`-funktiota, saa oman kopionsa `f`-funktion koodista ja `f`-funktion staattisista muuttujista! (Asiaa huonontaa vielä se, että `f`-funktion jokainen kopio ja jokainen kopio `f`-funktion staattisista muuttujista tulee päätymään näennäismuistin eri sivuille, joten kaksi kutsua `f`-funktion eri kopioihin tulee aiheuttamaan yhden tai useamman sivuvirheen.)

Ja lisää seuraa. Köyhien, taistelujärjestyksessä olevien kääntäjiesi täytyy joskus luoda funktiorunko avoimelle funktiolle myös silloin, kun ne ovat täysin halukkaita tekemään funktiosta avoimen. Varsinkin silloin, jos ohjelmasi ottaa avoimen funktion osoitteen, kääntäjien täytyy luoda funktiorunko sille. Kuinka voidaan saada aikaiseksi osoitin funktioon, jota ei ole olemassa?

```
inline void f() {...}          // sama kuin edellä

void (*pf)() = f;              // pf osoittaa f-funktioon

int main()
{
  f();                         // inline-tyyp. kutsu f

  pf();                        // ei-inline-tyyp. kutsu
                               // f-funktioon pf:n avulla
  ...
}
```

Päädyt tässä tapauksessa näennäisen paradoksaaliseen tilanteeseen, jossa `f`-funktioon tapahtuvat kutsut ovat avoimia funktioita, mutta - vanhojen sääntöjen vallitessa - jokainen käännösyksikkö, joka ottaa `f`-funktion osoitteen, luo silti staattisen kopion funktiosta. (Kun uudet säännöt pätevät, `f`-funktiosta luodaan vain yksi rivin ulkopuolinen kopio, riippumatta mukana olevien käännösyksiköiden määrästä.)

Tämä näkökulma suljetuista avoimista funktioista voi vaikuttaa sinuun vaikka et koskaan käyttäisi osoittimia funktioihin, koska ohjelmoijat eivät välttämättä ole ainoita, jotka pyytävät osoittimia funktioihin. Kääntäjät pyytävät joskus osoittimia funktioihin. Erityisesti silloin, kun kääntäjät luovat joskus rivin ulkopuolisia kopioita muodostinfunktioista ja tuhoajafunktioista niin, että ne voivat saada osoittimia näihin funktioihin, jotka ovat käytössä taulukossa olevien luokan olioiden muodostamiseen ja tuhoamiseen.

Muodostinfunktiot ja tuhoajafunktiot ovat itse asiassa usein huonompia ehdokkaita avoimien funktioiden muodostamiseen kuin satunnainen tarkastus osoittaisi. Esimerkiksi tutki alla olevan luokan `Derived` muodostinfunktiota:

```
class Base {
public:
  ...
private:
  string bm1, bm2;             // kantajäsenet 1 ja 2
};

class Derived: public Base {
public:
  Derived() {}                 // Derived-luokan mtin on
  ...                          // tyhjä – vai onko se?
```

```
private:
  string dm1, dm2, dm3;         // periytetyt jäsenet 1-3
};
```

Tämä muodostinfunktio näyttää varmasti erinomaiselta ehdokkaalta avointen funktioiden muodostamista varten, koska se ei sisällä koodia. Ulkonäkö voi kuitenkin pettää. Se, että se ei sisällä koodia, ei välttämättä tarkoita sitä, että se ei sisällä koodia. Se voi itse asiassa sisältää aikamoisen määrän koodia.

C++-kieli menee takuuseen eri asioista, joita tapahtuu, kun olioita luodaan ja tuhotaan. Kohdassa 5 kuvataan, kuinka silloin, kun käytät `new`-operaattoria, muodostinfunktiot alustavat dynaamisesti luodut funktiosi automaattisesti, ja kuinka silloin, kun käytät `delete`-operaattoria, vastaavia tuhoajafunktioita pyydetään avuksi. Kohdassa 13 selitetään, että kun luot olion, tuon olion jokainen kantaluokka ja jokainen tietojäsen muodostetaan automaattisesti, ja käänteinen tuhoamista koskeva prosessi tapahtuu automaattisesti silloin, kun olio tuhotaan. Nämä kohdat kuvaavat sitä, mitä C++:n mukaan täytyy tapahtua, mutta C++ ei kerro, *kuinka* asiat tapahtuvat. Tämä riippuu kääntäjän työkaluista, mutta pitäisi olla selvää, että tällaiset asiat eivät tapahdu itsestään. Ohjelmassasi täytyy olla koodia, joka saa nämä asiat tapahtumaan, ja tuon koodin - joka on kirjoitettu kääntäjän työkaluilla ja lisätty ohjelmaasi kääntämisen aikana - täytyy sijoittua jonnekin. Joskus se päätyy muodostinfunktioihin ja tuhoajafunktioihin, joten eräät toteutukset luovat koodia, joka on vastaavaa kuin alla olevalle muka tyhjälle `Derived`-muodostinfunktiolle luotu:

```
// Derived-muodostinfunktion mahdollinen toteutus
 Derived::Derived()
{
  // varaa kekomuistia tälle oliolle, jos sen on tarkoitus
  // olla keossa; katso Kohdasta 8 tietoa operator new
  // -funktiosta
  if (tämä olio on keossa)
    this = ::operator new(sizeof(Derived));

  Base::Base();                 // alusta Base-osa

  dm1.string();                 // muodosta dm1
  dm2.string();                 // muodosta dm2
  dm3.string();                 // muodosta dm3
}
```

Tämänkaltaisen koodin ei koskaan odoteta kääntyvän, koska se ei ole sallittua C++-koodia - ei ainakaan sinulle. Sinulla ei ensinnäkään ole mitään keinoa kysyä sen muodostinfunktion sisältä onko olio keossa. Toiseksi `this`-muuttujaan sijoittaminen on kielletty. Etkä voi myöskään pyytää muodostinfunktioita avuksi funktiokutsujen kautta. Kääntäjäsi toimivat kuitenkin huolimatta näistä rajoituksista - ne voivat tehdä aivan mitä haluavat. Mutta tämän lähdekoodin laillisuus ei ole pääasia. Pääasia on, että koodi, jolla kutsutaan `operator new` -funktiota (jos tarpeellista), muodostetaan

kantaluokan osat sekä tietojäsenet, voidaan hiljaisesti lisätä muodostinfunktioihisi. Kun näin on, nämä muodostinfunktiot kasvavat kooltaan, tehden näistä täten vähemmän puoleensavetäviä ehdokkaita avoimien funktioiden muodostamista varten. Sama järkeily pätee tietysti `Base`-muodostinfunktioon, joten jos se on muodostettu avoimena funktiona, kaikki siihen lisätty koodi lisätään myös `Derived`-muodostinfunktioon (`Derived`-muodostinfunktion kutsun kautta `Base`-muodostinfunktioon). Ja jos `string`-muodostinfunktio sattuu myös olemaan avoin funktio, `Derived`-muodostinfunktio hankkii *viisi kopiota* tuon funktion lähdekoodista, yhden jokaiselle `Derived`-olion viidestä merkkijonosta (kaksi, jotka se perii sekä kolme sen itse esittelemää). Näetkö nyt, miksi ei välttämättä ole aivan järjetöntä tehdä `Derived`-olion muodostinfunktiosta avoin funktio? Sama harkinta pätee tietysti `Derived`-olion tuhoajafunktioon, jonka täytyy tavalla tai toisella varmistaa, että kaikki `Derived`-olion muodostinfunktion alustamat oliot on tuhottu kunnolla. Sen täytyy myös ehkä vapauttaa dynaamisesti varattua muistia, jonka juuri tuhottu `Derived`-olio oli aikaisemmin vallannut.

Kirjastojen suunnittelijoiden täytyy arvottaa funktioiden esitteleminen `inline`-tyyppisinä, koska avoimet funktiot tekevät mahdolliseksi tarjota binäärisiä päivityksiä kirjastojen avoimiin funktioihin. Toisin sanoen, jos `f` on avoin funktio kirjastossa, kirjastojen asiakkaat kääntävät `f`-funktion rungon sovelluksiinsa. Jos kirjaston toteuttaja päättää myöhemmin muuttaa `f`-funktion, kaikki asiakkaat, jotka ovat käyttäneet sitä, täytyy kääntää uudelleen. Tämä ei ole kovin usein toivottavaa (katso myös Kohta 34). Toisaalta, jos `f` ei ole avoin funktio, `f`-funktion muutos vaatii ainoastaan sen, että asiakkaat linkittävät uudelleen. Tämä on selvästi vähemmän hankala rasitus kuin uudelleenkääntäminen ja, jos kirjasto sisältää dynaamisesti linkitetyn funktion, se voidaan upottaa sillä tavalla, että se on täysin näkymätön asiakkaille.

Nämä kaikki harkinnat on syytä pitää mielessä ohjelmistokehityksessä, mutta puhtaasti käytännöllisesti katsoen, koodauksen aikana on yksi asia, joka dominoi kaikkia muita: useimmilla debuggereilla on hankaluuksia avointen funktioiden kanssa.

Tämän ei pitäisi olla mikään suuri paljastus. Kuinka asetat pysähdyskohdan funktioon, joka ei ole kirjastossa? Kuinka käyt askel askeleelta läpi tällaisen funktion? Kuinka metsästät siihen tapahtuvat kutsut? Olematta mitenkään kohtuuttoman ovela (tai kierosti salakähmäinen), et yksinkertaisesti voi. Tämä johtaa onnellisesti loogiseen strategiaan, jossa päätetään, mitkä funktiot pitäisi esitellä avoimina ja mitkä ei.

Älä aluksi muodosta mitään avoimia funktioita, tai ainakin rajoita avoimuus niihin funktioihin, jotka ovat vilpittömästi vähämerkityksellisiä, kuten alla oleva `age`:

```
class Person {
public:
  int age() const { return personAge; }

  ...

private:
  int personAge;

  ...

};
```

Kun otat avoimet funktiot käyttöön varovaisesti, helpotat debuggereiden käyttöä, mutta sijoitat myös avointen funktioiden muodostamisen oikealle paikalleen: optimointina, joka on otettu käsin käyttöön. Älä unohda empiirisesti määritettyä sääntöä 80-20, joka korostaa, että tyypillinen ohjelma käyttää 80 prosenttia ajastaan suorittaen vain 20 prosenttia koodistaan. Se on tärkeä sääntö, koska se muistuttaa sinua siitä, että tavoitteesi ohjelmistokehittäjänä on tunnistaa se 20 prosenttia koodistasi, joka todella pystyy kasvattamaan ohjelmasi kokonaissuoritusta. Voit tehdä funktioistasi avoimia ja muutenkin säätää funktioitasi, kunnes lehmät tulevat kotiin, mutta olet ponnistellut turhaan, paitsi jos olet keskittymässä *oikeisiin* funktioihin.

Kun olet tunnistanut sovelluksesi tärkeiden funktioiden joukon, eli ne, joissa avointen funktioiden muodostamisella on todella merkitystä (joukko, joka itse on riippuvainen käyttämästäsi arkkitehtuurista), älä epäröi määrittää niitä `inline`-tyyppisinä. Varaudu kuitenkin samaan aikaan koodipöhöttymän aiheuttamiin ongelmiin, ja tarkkaile kääntäjän varoituksia (katso Kohta 48), jotka osoittavat, että avoimia funktioitasi ei ole muodostettu avoimina funktioina.

Kun avoimia funktioita käytetään arvostelukykyisesti, ne ovat arvaamattoman arvokas komponentti jokaisen C++-ohjelmoijan työkalupakissa, mutta kuten edelläkäyty keskustelu on paljastanut, niiden käyttö ei ole niin yksinkertaista ja suoraviivaista, kuin olisit voinut luulla.

Kohta 34: Minimoi kääntämisen riippuvaisuutta tiedostojen välillä.

Avaat siis C++-ohjelmasi ja teet pienen muutoksen luokan toteutukseen. Ota huomioon, että ei luokan rajapintaan, vaan pelkästään toteutukseen; vain yksityiseen osaan. Valmistaudut sitten rakentamaan ohjelmasi uudelleen arvioiden, että kääntäminen ja linkittäminen kestävät vain muutaman sekunnin. Olethan muuttanut vain yhtä luokkaa. Napsautat Rebuild-painiketta tai kirjoitat `make` (tai sen moraalisen vastineen), ja olet hämmästynyt, sitten murtunut, kun huomaat, että koko *maailma* käännetään ja linkitetään uudelleen!

Tämä kaikkihan saa sinut todella *vihaiseksi*?

Ongelma on siinä, että se, mitä C++ ei tee kovin hyvin, on rajapintojen erottaminen toteutuksesta. Varsinkaan luokkien määritykset eivät sisällä pelkästään rajapinnan määrityksiä, vaan ne sisältävät myös koko joukon toteutuksen yksityiskohtia. Esimerkiksi:

```
class Person {
public:
  Person(const string& name, const Date& birthday,
         const Address& addr, const Country& country);
  virtual ~Person();

  ...                    // kopiomuodostin ja sijoitus-
                         // operaattori jätetty pois
                         // yksinkertaisuuden vuoksi
  string name() const;
  string birthDate() const;
  string address() const;
  string nationality() const;

private:
  string name_;                 // toteutusyksityiskohta
  Date birthDate_;              // toteutusyksityiskohta
  Address address_;             // toteutusyksityiskohta
  Country citizenship_;         // toteutusyksityiskohta
};
```

Tämä tuskin on luokkasuunnittelua, joka voittaa Nobelin palkinnon, vaikka se havainnollistaa kiinnostavan nimeämiskäytännön, jolla erotetaan yleinen tieto julkisista funktioista silloin, kun samat nimet ovat järkeviä molemmille: jälkimmäiset on merkitty lopussa olevalla alaviivamerkillä. Kannattaa huomioida se tärkeä asia, että `Person`-luokkaa ei voida kääntää, paitsi jos kääntäjällä ei ole pääsyä luokan määrityksiin `Person`-luokan toteutuksen ehdoilla, nimittäin `string`, `Date`, `Address` ja `Country`. Tällaiset määritykset on tyypillisesti säädetty `#include`-direktiiveillä, joten tulet melko varmasti löytämään tämänkaltaista koodia tiedoston yläosassa, jossa määritetään `Person`-luokka:

```
#include <string>       // tietoa tyypin merkkijonosta
#include "date.h"       // (katso Kohta 49)
#include "address.h"
#include "country.h"
```

Valitettavasti tämä auttaa perustamaan käännöksestä riippuvuuden sen tiedoston, joka määrittelee `Person`-luokan, ja näiden sisällytettyjen tiedostojen välillä. Tästä on tuloksena, että jos mikään näistä tukiluokista muuttaa toteutustaan, tai jos mikään luokka, josta se on riippuvainen, muuttaa *omaa* toteutustaan, se tiedosto, joka sisältää `Person`-luokan, täytyy kääntää uudelleen, samoin kuin ne tiedostot, jotka käyttävät `Person`-luokkaa. Tämä voi olla enemmän kuin harmittavaa `Person`-luokan asiakkaille. Se voi olla suorastaan lamauttavaa.

Ihmettelet varmaan, miksi C++ vaatii luokan toteutuksen yksityiskohtien sijoittamista luokan määritykseen. Miksi et voi määrittää `Person`-luokkaa esimerkiksi tällä tavalla:

```
class string;    // "käsitteellinen" merkkijonotyypin
                 // esittely. Kts. Kohdasta 49 yks.kohdat

class Date;      // esittely eteenpäin
class Address;   // esittely eteenpäin
class Country;   // esittely eteenpäin

class Person {
public:
  Person(const string& name, const Date& birthday,
         const Address& addr, const Country& country);
  virtual ~Person();

  ...                            // kopiomuod., operator=

  string name() const;
  string birthDate() const;
  string address() const;
  string nationality() const;
};
```

määrittäen luokan toteutuksen yksityiskohdat erikseen? Jos näin olisi mahdollista, `Person`-luokan asiakkaiden täytyisi kääntää uudelleen vain, jos luokan rajapinta muuttuisi. Koska rajapinnoilla on taipumus vakiintua ennen toteutuksia, tällainen toteutuksen erotus rajapinnasta voisi säästää lukemattomia uudelleen kääntämisen ja linkittämisen tunteja laajan ohjelmistoponnistuksen aikana.

Kova maailma tunkeutuu valitettavasti tähän idylliseen skenaarioon, jota tulet varmasti arvostamaan, kun tutkit jotain tämänkaltaista:

```
int main()
{
  int x;                         // määritä int-tyyppi

  Person p(...);                 // määritä Person-luokka
                                 // (argumentit jätetty pois
  ...                            // yksinkertais. vuoksi)

}
```

Kun kääntäjät näkevät `x`-muuttujan määrityksen, ne tietävät, että niiden täytyy varata tarpeeksi muistia tallentamaan `int`-tyyppisen arvon. Ei ongelmaa. Jokainen kääntäjä tietää, kuinka suuri `int`-tyyppinen luku on. Kun kääntäjät kuitenkin näkevät `p`:n määrityksen, ne tietävät, että niiden täytyy varata tarpeeksi tilaa `Person`-luokalle, mutta kuinka niiden oletetaan tietävän, kuinka iso `Person`-olio on? Tämä tieto voidaan saada selville vain luokan määrityksen avulla. Jos luokan määritykselle olisi

sallittua olla mainitsematta toteutuksen yksityiskohtia, kuinka kääntäjät tietäisivät, kuinka paljon tilaa tulisi varata muistista?

Tämä ei periaatteessa ole ylipääsemätön ongelma. Kielet, kuten Smalltalk, Eiffel ja Java, väistelevät tätä koko ajan. Ne tekevät sen varata vain tarpeeksi tilaa olion *osoittimelle* silloin, kun olio määritetään. Tämä tarkoittaa, että ne hoitavat edellä mainitun koodin aivan kuin se olisi kirjoitettu tällä lailla:

```
int main()
{
  int x;                        // määritä int-tyyppi
  Person *p;                    // määritä osoitin
                                // Person-luokkaan
  ...
}
```

Sinulle on ehkä juolahtaa mieleen, että tämä on itse asiassa laillista C++-kielessä, ja ilmenee, että pelaat itsekin peliä nimeltä "piilota olion toteutus osoittimen taakse".

Tässä on tapa, jolla otat käyttöösi tekniikan, jolla irrotat `Person`-luokan rajapinnan toteutuksestaan. Kirjoitat ensin vain seuraavan otsikkotiedostoon, jossa esitellään `Person`-luokka:

```
// kääntäjien täytyy silti tietää näistä
// Person-muodostimen tyyppinimistä
class string;     // katso jälleen Kohta 49 miksi
                  // tämä ei ole oikein merkkijonolle
class Date;
class Address;
class Country;

// PersonImpl-luokka sisältää Person-luokan
// toteutuksen yksityiskohdat; tämä on vain luokan nimen
// etukäteen määritys
class PersonImpl;

class Person {
public:
  Person(const string& name, const Date& birthday,
         const Address& addr, const Country& country);
  virtual ~Person();

  ...                           // kopiomuod., operator=

  string name() const;
  string birthDate() const;
  string address() const;
  string nationality() const;

private:
  PersonImpl *impl;             // osoitin toteutukseen
};
```

`Person`-luokan asiakkaat ovat nyt täysin erotettuja yksityiskohdista, jotka koostuvat merkkijonoista, päivämääristä, osoitteista, maista ja henkilöistä. Nämä luokat ovat tarvittaessa muutettavissa, mutta `Person`-luokan asiakkaat voivat edelleen olla autuaasti tietämättömiä. Yksityiskohtaisemmin, he voivat olla autuaasti tietämättömiä ilman uudelleen kääntämistä. Lisäksi, koska eivät voi nähdä `Person`-luokan toteutuksen yksityiskohtia, asiakkaat tuskin tulevat kirjoittamaan koodia, joka jotenkin riippuu näistä yksityiskohdista. Kyseessä on rajapinnan ja toteutuksen todellinen erottaminen.

Avain tähän erottamiseen on riippuvaisuuksien korvaaminen *luokkamäärityksillä*, joilla on riippuvaisuuksia luokan *esittelyihin*. Sinun tarvitsee tietää vain tämä kääntämisen riippuvuuksien minimoinnista: tee otsikkotiedostoistasi omavaraisia aina, kun se on käytännöllistä, ja silloin kun se ei ole käytännöllistä, ole riippuvainen luokan esittelyistä, älä luokan määrityksistä. Kaikki muu sujuu itsekseen tämän yksinkertaisen suunnittelustrategian jälkeen.

On olemassa kolme välitöntä implikaatiota:

- **Vältä olioiden käyttämistä silloin, kun olioiden viittaukset ja osoittimet riittävät**. Voit määrittää tyypin viittaukset ja osoittimet pelkästään tyypin *esittelyllä*. Tyypin *olioiden* määrittäminen tekee välttämättömäksi tyypin määrityksen läsnäolon.
- **Käytä aina kun voit luokan esittelyitä luokan määritysten sijasta**. Huomaa, että et *koskaan* tarvitse luokan määritystä esittelemään funktiota, joka käyttää tuota luokkaa, et edes, vaikka funktio välittää tai palauttaa luokan tyypin arvoparametrillä:

  ```
  class Date;                    // luokkamääritys

  Date returnADate();            // toimii – Date-luokan
  void takeADate(Date d);        // määritystä ei tarvita
  ```

  Arvoparametrillä välittäminen on tietysti yleisesti ottaen huono ajatus (katso Kohta 22), mutta jos huomaat, että olet pakotettu käyttämään sitä syystä tai toisesta, kääntäjän tarpeettomien riippuvaisuuksien esittelyyn ei silti ole oikeutusta.

  Jos olet yllättynyt, että `returnADate-` ja `takeADate`-esittelyt kääntyvät ilman `Date`-luokan määritystä, liity kerhoon; niin olin minäkin. Se ei kuitenkaan ole niin kummallista kuin miltä se näyttää, koska jos joku *kutsuu* näitä funktioita, `Date`:n määrityksen täytyy olla näkyvillä. Tiedän mitä ajattelet: miksi vaivautua esittelemään funktioita, joita kukaan ei kuitenkaan kutsu? Yksinkertaista. Asia ei ole niin, että *kukaan* ei kutsu niitä, vaan kaikki eivät kutsu niitä. Jos sinulla esi-merkiksi on kirjasto, joka sisältää satoja funktioiden esittelyjä (jotka ovat mahdollisesti jakautuneet useisiin nimiavaruuksiin - katso Kohta 28), on epätodennäköistä, että jokainen asiakas kutsuu jokaista funktiota. Kun poistat sen taakan, että tarjoat luokan määritykset (`#include`-direktiivien avulla)

otsikkotiedostosta, joka koostuu funktion *esittelyistä* asiakkaiden tiedostoihin, jotka sisältävät *funktiokutsut*, eliminoit asiakkaan keinotekoiset riippuvuudet tyyppimäärityksistä, joita he eivät tarvitse todella.

- **Älä sisällytä otsikkotiedostoja otsikkotiedostoissasi `#include`-direktiiveillä, paitsi jos otsikkosi eivät käänny ilman niitä.** Esittele tarvitsemasi luokat sen sijaan käsin, anna otsikkotiedostojesi asiakkaiden sisällyttää `#include`-direktiivillä tarvittavat lisäotsikot, jotta *heidän* lähdekoodinsa kääntyisi. Jotkut asiakkaat voivat murahtaa, että tämä on epämukavaa, mutta voit levätä huojentuneena, että säästät heitä suuremmalta tuskalta, kuin mitä aiheutat. Tätä tekniikkaa pidetään itse asiassa niin hyvänä, että se sisältyy perus-C++-kirjastoon (katso Kohta 49); otsikko `<iosfwd>` sisältää *esittelyt* (ja vain esittelyt) iostream-kirjaston tyypeille.

Luokkia, kuten `Person`, jotka sisältävät vain osoittimen täsmentämättömään toteutukseen, kutsutaan usein *Handle*-luokiksi tai *Envelope*-luokiksi. (Aiemmassa tapauksessa luokkia, joihin ne osoittavat, kutsutaan usein *Body*-luokiksi; jälkimmäisessä tapauksessa osoitetut luokat tunnetaan *Letter*-luokkina.) Voit toisinaan kuulla ihmisten viittaavan luokkiin kuten *Cheshire Cat*, viittaus *Liisa Ihmemaassa* -kirjan kissaan, joka pystyi halutessaan jättämään taakseen vain hymyn sen jälkeen, kun muu oli kadonnut.

Jottet ihmettelisi, tekevätkö Handle-luokat ylipäätään mitään, vastaus on yksinkertainen: ne välittävät kaikki funktiokutsusi eteenpäin vastaaville Body-luokille, ja nuo luokat tekevät varsinaisen työn. Tässä on esimerkiksi tapa, jolla kaksi `Person`-luokan jäsenfunktiota voitaisiin toteuttaa:

```
#include "Person.h"       // koska olemme toteuttamassa
                          // Person-luokkaa, meidän täytyy
                          // sisällyttää
                          // (#include) luokkamääritys

#include "PersonImpl.h"   // meidän täytyy myös sisällyt-
                          // tää #include-komennolla
                          // PersonImpl:n luokkamääritys,
                          // emme muuten voisi kutsua
                          // sen jäsenfunktioita. Huomaa,
                          // että PersonImpl-luokalla on
                          // samat jäsenfunktiot kuin
                          // Person-luokalla - niiden
                          // rajapinnat ovat identtiset

Person::Person(const string& name, const Date& birthday,
               const Address& addr, const Country& country)
{
  impl = new PersonImpl(name, birthday, addr, country);
}
```

```
string Person::name() const
{
  return impl->name();
}
```

Huomaa, kuinka `Person`-muodostinfunktio kutsuu `PersonImpl`-muodostinfunktiota (implisiittisesti, käyttäen `new`-operaattoria - katso Kohta 5) ja kuinka `Person::name` kutsuu `PersonImpl::name`-funktiota. Tämä on tärkeää. Se, että `Person`-luokasta tehdään Handle-tyyppinen luokka, ei muuta sitä, mitä `Person` tekee, se vain muuttaa sen, missä se tekee sen.

Vaihtoehto Handle-luokka-työtavalle on tehdä `Person`-luokasta erikoistyyppinen abstrakti kantaluokka nimeltä *Protocol class*. Protocol-luokalla ei määrityksenä ole toteutusta; sen syy olemiseen (*raison d'être*) on määrittää rajapinta periytetyille luokille (katso Kohta 36). Tästä on tuloksena, että sillä ei tyypillisesti ole tietojäseniä, muodostinfunktioita, virtuaalituhoajafunktioita (katso Kohta 14) ja joukkoa puhtaita virtuaalifunktioita, jotka määrittelevät rajapinnan. `Person`-luokan Protocol-luokka voisi näyttää tältä:

```
class Person {
public:
  virtual ~Person();

  virtual string name() const = 0;
  virtual string birthDate() const = 0;
  virtual string address() const = 0;
  virtual string nationality() const = 0;
};
```

`Person`-luokan asiakkaiden täytyy ohjelmoida `Person`-luokan osoittimien ja viittausten ehdoilla, koska luokkia, jotka sisältävät puhtaita virtuaalifunktioita, ei ole mahdollista instantioida. (`Person`-luokasta *periytettyjä* luokkia voidaan kuitenkin instantioida - katso alla.) Kuten Handle-luokan asiakkaita, Protocol-luokkien asiakkaiden ei tarvitse kääntää uudelleen, paitsi jos Protocol-luokan rajapintaa on muutettu.

Protocol-luokan asiakkailla täytyy tietenkin olla *jokin* tapa luoda uusia olioita. Ne tekevät sen tyypillisesti kutsumalla funktiota, joka on muodostinfunktion roolissa piilotetuille (periytetyille) luokille, jotka on itse asiassa instantioitu. Näitä funktioita kutsutaan monilla eri nimillä (muun muassa tehdasfunktiot (*factory functions*) ja virtuaaliset muodostinfunktiot (*virtual constructors*), mutta ne kaikki käyttäytyvät samalla tavalla: ne palauttavat osoittimia dynaamisesti varattuihin olioihin, jotka tukevat Protocol-luokan rajapintaa. Tämänkaltainen funktio voitaisiin esitellä seuraavasti:

```
// makePerson on "virtuaalimuodostin" (eli "tehdas-
// funktio") olioille, jotka tukevat Person-rajapintaa
Person*
  makePerson( const string& name,       // palauta osoitin
              const Date& birthday,     // uuteen Person-l.
              const Address& addr,      // joka on alustet.
              const Country& country);  // ann. parametr.
```

ja asiakkaat käyttävät sitä tällä lailla:

```
string name;
Date dateOfBirth;
Address address;
Country nation;

...

// luo olio, joka tukee Person-rajapintaa
Person *pp = makePerson(name, dateOfBirth, address, nation);

...

cout << pp->name()              // käytä oliota Person-
     << " was born on "         // luokan rajapinn. kautta
     << pp->birthDate()
     << " and now lives at "
     << pp->address();

...

delete pp;                      // poista olio sitten, kun
                                // sitä ei enää tarvita
```

Koska `makePerson`-funktion kaltaiset funktiot liittyvät läheisesti Protocol-luokkaan, jonka rajapintaa niiden luomat oliot tukevat, on tyylikästä esitellä ne Protocol-luokan sisällä `static`-tyyppisenä:

```
class Person {
public:
  ...                           // sama kuin edellä

// makePerson on nyt luokan jäsen
static Person * makePerson( const string& name,
                            const Date& birthday,
                            const Address& addr,
                            const Country& country);
};
```

Tällä vältetään globaalin nimiavaruuden (tai minkä tahansa nimiavaruuden) tukkiminen useilla tämän tyyppisillä funktioilla (katso Kohta 28).

Jossain vaiheessa täytyy tietenkin määrittää varsinaisia todellisia luokkia, jotka tukevat Protocol-luokan rajapintaa, ja todellisia muodostinfunktioita täytyy kutsua. Tämä kaikki tapahtuu näyttämön takana virtuaalimuodostinfunktioiden toteutustiedostojen sisällä. Protocol-luokalla `Person` voisi esimerkiksi olla todellinen periytetty luokka `RealPerson`, joka sisältää toteutukset sen perimille virtuaalifunktioille:

```
class RealPerson: public Person {
public:
  RealPerson(const string& name, const Date& birthday,
             const Address& addr, const Country& country)
  : name_(name), birthday_(birthday),
    address_(addr), country_(country)
  {}
```

```
  virtual ~RealPerson() {}

    string name() const;           // näiden funktioiden
    string birthDate() const;      // toteutuksia ei
    string address() const;        // näytetä, mutta ne on
    string nationality() const;    // helppo kuvitella

  private:
    string name_;
    Date birthday_;
    Address address_;
    Country country_;
  };
```

Tälle `RealPerson`-luokalle on todella tarpeetonta kirjoittaa `Person::makePerson`-funktiota:

```
  Person * Person::makePerson(const string& name,
                              const Date& birthday,
                              const Address& addr,
                              const Country& country)
  {
    return new RealPerson(name, birthday, addr, country);
  }
```

`RealPerson` esittää toisen kahdesta yleisimmästä mekanismista, jolla luokka Protocol toteutetaan: se perii rajapintansa määrityksen Protocol-luokasta (`Person`), se toteuttaa sitten funktiot rajapinnassa. Toinen tapa, jolla Protocol-luokka toteutetaan, liittyy moniperintään, aiheeseen, jota käsitellään Kohdassa 43.

Hyvä on, Handle-luokat ja Protocol-luokat siis irrottavat rajapinnat toteutuksista, vähentäen täten riippuvuutta kääntämisestä tiedostojen välillä. Kyynikko kun olet, niin tiedän, että odotat jotain todella hämärää. Mutiset, että "Mitä nämä kaikki taikatemput maksavat minulle?" Vastaus on se tavallinen tietojenkäsittelyopin vastaus: hintana on vaikutus nopeuteen suorituksen aikana, sekä hieman lisämuistia per olio.

Jäsenfunktioiden täytyy Handle-luokkien tapauksessa kulkea toteutusosoittimen läpi päästäkseen olion tietoon. Tämä lisää yhden tason mutkia jokaista pääsyä kohti. Ja tämän toteutusosoittimen kokoa täytyy lisätä siihen muistin määrään, joka vaaditaan tallentamaan jokainen olio. Toteutusosoitin täytyy lopulta alustaa (Handle-luokan muodostinfunktioilla) osoittamaan dynaamisesti varattuun toteutusolioon, joten sinulle koituvat kustannukset, jotka ovat periytyviä dynaamisessa muistin varauksessa (ja myöhemmin muistin vapauttamisessa) - katso Kohta 10.

Jokainen funktiokutsu on virtuaalinen Protocol-luokille, joten maksat epäsuorasta hypystä muodostuvat kustannukset joka kerta, kun teet funktiokutsun (katso Kohta 14). Protocol-luokasta periytettyjen olioiden täytyy lisäksi sisältää osoitin virtuaaliseen tauluun (katso jälleen Kohta 14). Tämä osoitin voi kasvattaa olion tallentamiseen tar-

vittavan muistin määrää riippuen siitä, onko Protocol-luokka muut poissulkeva virtuaalifunktioiden lähde oliolle.

Lopuksi, Handle-luokat ja Protocol-luokat eivät kumpikaan saa paljoa irti avoimista funktioista. Kaikki avointen funktioiden käytännöllinen käyttö vaatii pääsyn toteutuksen yksityiskohtiin, ja tätähän varten Handle- ja Protocol-luokat alkujaan suunniteltiin välttämään.

Olisi kuitenkin vakava virhe irtisanoa Handle-luokat ja Protocol-luokat yksinkertaisesti siksi, että niihin liittyy kustannuksia. Liittyyhän virtuaalifunktioihinkin, etkä varmasti jättäisi niitä heitteille, ethän? (Jos tekisit, luet silloin väärää kirjaa.) Harkitse sen sijaan näiden tekniikoiden käyttämistä kehitysopilliseen tapaan. Käytä Handle- ja Protocol-luokkia kehityksen aikana vähentääksesi vaikutusta asiakkaisiin silloin kun toteutukset muuttuvat. Korvaa Handle- ja Protocol-luokat todellisilla luokilla tuotantokäyttöön silloin, kun voidaan osoittaa, että ero nopeudessa ja/tai koossa on tarpeeksi merkitsevä oikeuttamaan luokkien välillä kasvaneen kytkennän. Voimme vain toivoa, että jonain päivänä on tarjolla työkaluja, joilla tämänkaltainen muuntaminen voidaan suorittaa automaattisesti.

Handle-luokkien, Protocol-luokkien ja todellisten luokkien taitava sekoittaminen sallii sinun kehittää ohjelmistojärjestelmiä, jotka suoriutuvat tehokkaasti ja joita on helppo kehittää, mutta yksi vakava varjopuoli on silti: sinun täytyy ehkä lyhentää pitkiä taukoja, joita pidät silloin, kun käännät ohjelmasi uudelleen.