Luokat ja funktiot: suunnittelu ja esittely

Uuden luokan esittely ohjelmaan luo uuden tyypin: luokan suunnittelu on *tyypin* suunnittelua. Sinulla ei ehkä ole paljon kokemusta tyyppisuunnittelusta, koska useimmat kielet eivät tarjoa sinulle mahdollisuutta saada mitään harjoitusta. C++-kielessä se on ehdottoman tärkeää, ei pelkästään siksi, että voit suunnitella silloin kun haluat, vaan siksi, että *suunnittelet* joka kerta, kun esittelet luokan, tarkoitat sitä sitten tai et.

Hyvien luokkien suunnittelu on haastavaa, koska hyvien tyyppien suunnittelu on haastavaa. Hyvillä tyypeillä on luonnollinen kielioppi, intuitiivinen merkitysoppi, ja yksi tai useampi tehokas toteutustapa. Huonosti ajateltu luokkamääritys voi C++-kielessä tehdä mahdottomaksi saavuttaa yhtäkään näistä tavoitteista. Jopa luokan jäsenfunktion suorituksen erityistuntomerkit määritetään yhtä paljon näiden jäsenfunktioiden esittelyistä kuin niiden määrityksistä.

Kuinka sitten suunnitellaan tehokkaita luokkia? Ensinnäkin sinun täytyy ymmärtää pulmat, joita tulet kohtaamaan. Käytännöllisesti katsoen jokainen luokka vaatii sen, että kohtaat seuraavat kysymykset, joiden vastaukset johtavat usein suunnittelusi rajoituksiin:

- *Kuinka oliot pitäisi luoda ja tuhota?* Tapa, jolla tämä on tehty, vaikuttaa vahvasti muodostinfunktioiden ja tuhoajafunktioiden suunnitteluun, samoin omien `operator new`-, `operator new[]`-, `operator delete`- ja `operator delete[]` -funktioversioittesi suunnitteluun, jos kirjoitat ne.
- *Kuinka olion alustus eroaa olion sijoittamisesta?* Vastaus tähän kysymykseen määrittää muodostinfunktioiden ja sijoitusoperaattoreiden käyttäytymisen ja niiden väliset erot.
- *Mitä tarkoittaa välittää uuden tyyppisiä olioita arvoparametrillä?* Muista, että kopiomuodostin määrittää sen, mitä tarkoittaa välittää olio arvoparametrillä.

- *Mitkä ovat uuden tyypin lailliset rajoitukset?* Nämä rajoitukset määrittävät sen, minkä tyyppinen virheentarkistus sinun täytyy tehdä jäsenfunktioidesi sisälle, varsinkin muodostinfunktioiden ja sijoitusoperaattoreiden. Se voi myös vaikuttaa funktioidesi muodostamiin poikkeuksiin, ja jos käytät niitä, funktioidesi poikkeustilanteiden erittelyihin.
- *Sopiiko uusi tyyppi esitykseen periytyvyydestä?* Jos olemassaolevista luokista periytetään, sinua rajoittaa näiden luokkien suunnittelu, varsinkin se, ovatko periytetyt funktiot virtuaalisia vai eivät. Jos haluat muiden luokkien pystyvän periytymään luokastasi, se vaikuttaa siihen, ovatko esittelemäsi funktiot virtuaalisia.
- *Minkälaisia tyyppimuunnoksia sallitaan?* Jos haluat sallia A-tyypin olioiden tulevan *implisiittisesti* muutetuiksi B-tyyppisiksi olioiksi, sinun pitää kirjoittaa joko tyypin muuttamisen funktio luokassa A tai ei-`explicit`-tyyppinen muodostinfunktio luokassa B, jota voidaan kutsua yhdellä argumentilla. Jos haluat sallia vain *eksplisiittiset* muunnokset, sinun pitää kirjoittaa funktiot, jotka suorittavat muunnokset, mutta välttää tekemästä niitä tyypin muuttamisen operaattoreiksi tai ei-`explicit`-tyyppisiksi yhden argumentin muodostinfunktioiksi.
- *Mitkä operaattorit ja funktiot ovat järkeviä uudelle tyypille?* Vastaus tähän kysymykseen määrittää sen, mitä funktioita esittelet luokkasi rajapinnassa.
- *Minkä perusoperaattoreiden ja -funktioiden pitäisi olla eksplisiittisesti kiellettyjä?* Nämä sinun pitää määrittää arvolla `private`.
- *Kenellä pitäisi olla pääsy uuden tyypin jäseniin?* Tämän kysymyksen avulla sinun on helppo määrittää, mitkä jäsenet ovat julkisia, mitkä ovat suojattuja ja mitkä ovat yksityisiä. Se auttaa myös sinua määrittämään, minkä luokkien ja/tai funktioiden pitäisi olla ystäviä keskenään, samoin kuin onko järkevää pesiyttää luokka toisen sisälle.
- *Kuinka yleinen uusi tyyppi on?* Ehkä et olekaan määrittämässä uutta tyyppiä. Ehkä olet määrittämässä kokonaista tyyppi*perhettä*. Jos näin, niin sinun ei kannata määrittää uutta luokkaa, määritä uusi luokkamalli.

Näihin kysymyksiin on vaikea vastata, joten tehokkaiden luokkien määrittäminen C++-kielessä on kaukana yksinkertaisesta. Hyvin tehdyt, käyttäjän määrittelemät luokat C++-kielessä tuottavat kuitenkin tyyppejä, jotka ovat kaikkea muuta kuin erottumattomia sisäänrakennetuista tyypeistä, ja tämä tekee ponnistelusta kannattavan.

Jokaisesta yllä olevasta kysymyksestä käytävä keskustelu voisi ansaita oman kirjansa, joten seuraavat ohjeet ovat kaikkea muuta kuin kattavia. Ne korostavat kuitenkin suunnittelussa kaikkein tärkeimpiä ajattelun aiheita, varoittavat eräistä kaikkein yleisimmistä virheistä ja tarjoavat ratkaisuja yleisimpiin ongelmiin, joita luokan suunnittelijat kohtaavat. Monet ohjeet ovat sovellettavissa sekä ei-jäsen-funktioihin että jäsenfunktioihin, joten käsittelen tässä kappaleessa myös globaalien ja nimiavaruudessa olevien funktioiden suunnittelua ja esittelyä.

Kohta 18: Pyri luokan rajapintoihin, jotka ovat valmiita ja minimaalisia.

Asiakkaan rajapinta luokkaan on rajapinta, johon luokkaa käyttävien ohjelmoijien on helppo päästä. Tavallista on, että rajapinnassa on vain funktioita, koska jos asiakkaan rajapinnassa on tietojäseniä, siinä on omat haittansa (katso Kohta 20).

Saat harmaita hiuksia, kun yrität hahmottaa, mitä funktioita luokan rajapinnassa pitäisi olla. Kaksi täysin erilaista suuntaa on valittavissa. Haluaisit toisaalta rakentaa luokan, jota on helppo ymmärtää, suoraviivaista käyttää ja helppo toteuttaa. Tämä koskee yleensä kohtalaisen pientä määrää jäsenfunktioita, joista jokainen suorittaa erityisen tehtävän. Haluaisit toisaalta luokastasi tehokkaan ja mukavan käyttää, mikä usein tarkoittaa sitä, että joudut lisäämään funktioita sisällyttäen tuen usein suoritetuille tehtäville. Kuinka päätät sen, mitkä funktiot menevät luokkaan ja mitkä eivät?

Yritä tätä: pyri luokan rajapintaan, joka on *valmis* ja *minimaalinen*.

Valmis rajapinta sallii asiakkaiden tehdä kaikkea, mitä he kohtuuden nimessä voisivat haluta tehdä. Tämä tarkoittaa, että jokaiselle järkevälle tehtävälle, jonka asiakkaat haluaisivat suorittaa, olisi järkevä tapa suorittaa se, vaikka se ei olisikaan niin mukava, kuin asiakkaat haluaisivat sen olevan. *Minimaalinen* rajapinta sisältää toisaalta niin vähän funktioita kuin vain on mahdollista, ja kahdella jäsenfunktiolla ei olisi siinä päällekkäistä toiminnallisuutta. Jos tarjoat valmiin, minimaalisen rajapinnan, asiakkaat voivat tehdä mitä tahansa haluavatkin, mutta luokan rajapinta ei ole yhtään monimutkaisempi, kuin on ehdottomasti tarpeellista.

Valmiin rajapinnan toivottavuus tuntuu ilmeisen selvältä, mutta miksi minimaalinen rajapinta? Miksei annettaisi asiakkaille kaikkea mitä he pyytävät lisäten toiminnallisuutta kunnes kaikki ovat onnellisia?

Huomioimatta moraalista kysymystä - onko todellakin *oikein* hemmotella asiakkaitasi? - luokan rajapinnassa, joka on täynnä funktioita, on ehdottomasti teknisiä haittoja. Ensinnäkin, mitä enemmän rajapinnassa on funktioita, sitä vaikeampi mahdollisten asiakkaiden on sitä ymmärtää. Mitä vaikeampi heidän on sitä ymmärtää, sitä haluttomampia he ovat oppimaan käyttämään sitä. Luokka, jossa on 10 funktiota, näyttää

sävyisältä useimmista ihmisistä, mutta luokka, jossa on 100 funktiota, saa monet ohjelmoijat pakenemaan ja menemään piiloon. Kun laajennat luokkasi toimivuutta tekemällä siitä mahdollisimman puoleensavetävän, päädyt ehkä lopulta viemään ihmisiltä rohkeuden oppia sen käyttö.

Laaja rajapinta voi myös johtaa hämmennykseen. Oletetaan, että luot luokan, joka tukee tekoälyn tiedontarvetta. Yksi jäsenfunktiosi on nimeltään `think`, mutta huomaat myöhemmin, että jotkut ihmiset haluavat sen nimen olevan `ponder`, ja muut mieluummin `ruminate`. Tarjoat sovitelleen kaikki kolme funktiota, vaikka ne tekevät saman asian. Mieti luokkasi mahdollisen asiakkaan hämmennystä, kun hän yrittää selvittää asioita. Asiakas on kasvokkain kolmen eri funktion kanssa, joiden kaikkien pitäisi tehdä sama asia. Voiko tämä todella olla totta? Onko näiden kolmen välillä olemassa jokin hienoinen ero, ehkä tehokkuudessa tai yleisyydessä tai luotettavuudessa? Jos ei, miksi on kolme eri funktiota? Sen sijaan, että asiakas arvostaisi joustavuuttasi, hän melko varmasti ihmettelee, mitä ihmettä olit ajatellut/think (tai puntaroinut/ponder tai märehtinyt/ruminate).

Ison luokkarajapinnan toinen haitta on ylläpidettävyys. On yksinkertaisesti paljon vaikeampaa ylläpitää ja laajentaa luokkaa, jossa on monta funktiota, kuin luokkaa, jossa on vähän funktioita. On paljon vaikeampi välttää kaksinkertaista lähdetekstiä (ja siihen liittyviä kaksoisvirheitä), ja on myös paljon vaikeampaa ylläpitää rajapinnan yhtenäisyyttä. Se on myös paljon vaikeammin dokumentoitavissa.

Lopuksi, pitkien luokkien määritysten seurauksena on myös pitkät otsikkotiedostot. Koska otsikkotiedostot luetaan normaalisti joka kerta, kun ohjelma käännetään (katso Kohta 34), luokkamääritykset, jotka ovat pidemmät kuin tarpeellista, voivat hankkia itselleen mojovan rangaistuksen projektin elinkaaren aikana tapahtuvassa kääntämisajassa.

Vastikkeeton funktioiden lisääminen rajapintaan ei lyhyesti sanottuna tapahdu ilman kustannuksia, joten sinun pitää miettiä huolellisesti, aiheuttaako uuden funktion mukavuus (uusi funktio voidaan lisätä *vain* mukavuuden takia, jos rajapinta on jo valmis) lisäkustannukset monimutkaisuudessa, tajuttavuudessa, ylläpidettävyydessä ja kääntämisen nopeudessa.

Kohtuuttomaan synkkyyteen ei ole silti aihetta vaipua. On usein perustellumpaa tarjota enemmän kuin vähimmäismäärän sisältävän joukon funktioita. Jos yleisesti suoritettu tehtävä voidaan toteuttaa paljon tehokkaammin jäsenfunktiona, se voi hyvin oikeuttaa sen lisäyksen rajapintaan. Jos jäsenfunktion lisäys tekee luokasta selvästi helpomman käyttää, se voi olla vakuutena sen lisäämisestä luokkaan. Ja jos jäsenfunktion lisäys tulee todennäköisesti estämään asiakkaiden virheet, se on myös tehokas argumentti sen olemiseen osa rajapintaa.

Tutki konkreettista esimerkkiä: malli luokille, joka toteuttaa taulukot asiakkaan määrittelemillä ylä- ja alarajoilla ja joka tarjoaa vapaaehtoisen raja-arvojen tarkistuksen. Tällaisen taulukkomallin alku nähdään alla:

```
template<class T>
class Array {
public:
  enum BoundsCheckingStatus {NO_CHECK_BOUNDS = 0,
                             CHECK_BOUNDS = 1};

  Array(int lowBound, int highBound,
        BoundsCheckingStatus check = NO_CHECK_BOUNDS);

  Array(const Array& rhs);

  ~Array();

  Array& operator=(const Array& rhs);

private:
  int lBound, hBound;           // ala- ja yläraja

  vector<T> data;               // taulukon sisältö; katso
                                // Kohdasta 49
                                // tietoa vectorista

  BoundsCheckingStatus checkingBounds;
};
```

Tähän mennessä esitellyt jäsenfunktiot ovat niitä, jotka eivät pohjimmiltaan vaadi ajattelemista/`think` (tai puntarointia/`ponder` tai märehtimistä/`ruminate`). Muodostinfunktiosi sallii asiakkaiden määrittää jokaisen taulukon raja-arvot, kopiomuodostimen, sijoitusoperaattorin ja tuhoajafunktion. Olet tässä tapauksessa esitellyt tuhoajafunktion ei-virtuaaliseksi, mikä viittaa siihen, että tätä luokkaa ei tulla käyttämään kantaluokkana (katso Kohta 14).

Sijoitusoperaattorin esittely on itse asiassa vähemmän selväpiirteistä, kuin miltä se aluksi tuntuu. C++-kielen sisäänrakennetut taulukothan eivät salli sijoittamista, joten et varmaan myöskään halua sallia sitä `Array`-olioille (katso Kohta 27). Taulukkotyyppinen `vector`-malli (peruskirjastossa - katso Kohta 27) sallii toisaalta sijoittamisen `vector`-olioiden välillä. Seuraat tässä esimerkissä `vector`-olion mallia ja tämä päätös, kuten tulet näkemään, tulee vaikuttamaan luokan rajapinnan muihin osiin.

Vanhat C-sällit irvistäisivät nähdessään tämän rajapinnan. Missä on tuki määrätyn kokoisen taulukon esittelylle? Luulisi olevan helppoa lisätä toinen muodostinfunktio,

```
Array(int size,
      BoundsCheckingStatus check = NO_CHECK_BOUNDS);
```

mutta tämä ei ole osa minimaalista rajapintaa, koska muodostinfunktiota, joka ottaa ylemmän ja alemman raja-arvon, voidaan käyttää suorittamaan sama tehtävä. Voisi kuitenkin olla viisas poliittinen siirto kohdella lempeästi vanhoja sällejä, mahdollisesti kantakielen johdonmukaisuuden otsikon alaisuudessa.

Mitä muita funktioita tarvitaan? Osa valmista rajapintaa on varmasti hakemiston luominen taulukkoon:

```
// palauta elementti lukemista/kirjoittamista varten
T& operator[](int index);

// palauta vain lukuoikeudella var. elementti
const T& operator[](int index) const;
```

Kun esittelet saman funktion kahdesti, yhden kerran `const`-tyyppisenä ja toisen kerran ei-`const`-tyyppisenä, tarjoat tuen sekä `const`- että ei-`const`-tyyppisille `Array`-olioille. Ero paluuarvoissa on merkitsevä, tämä selvitetään Kohdassa 21.

`Array`-malli tukee nyt muodostamista, tuhoamista, parametrien välittämistä arvoparametreinä, sijoitusta ja indeksointia, ja tämä voi sinusta tuntua valmiilta rajapinnalta. Mutta katso tarkemmin. Oletetaan, että asiakas haluaa tutkia kokonaisluvuista koostuvaa taulukkoa silmukalla, tulostaen jokaisen sen elementin, tähän tyyliin:

```
Array<int> a(10, 20);          // a:n rajat ovat 10 - 20

...

for (int i = a:n alaraja; i <= a:n yläraja; ++i)
  cout << "a[" << i << "] = " << a[i] << '\n';
```

Kuinka asiakas saa `a`-muuttujan raja-arvot? Vastaus riippuu siitä, mitä tapahtuu `Array`-olioiden sijoituksen aikana, eli mitä tapahtuu `Array::operator=`-funktion sisällä. Jos sijoitus voi varsinkin muuttaa `Array`-olion raja-arvoja, sinun täytyy sisällyttää jäsenfunktiot, jotka palauttavat nykyiset raja-arvot, koska asiakkaalla ei ole mitään keinoa tietää ennalta, mitkä raja-arvot olivat ohjelman missäkin vaiheessa. Jos `a` oli edellä mainitussa esimerkissä sijoituksen kohde aikavälillä, jolla se määritettiin ja jolla sitä käytettiin silmukassa, asiakkaalla ei ole mitään keinoa määrittää `a`:n senhetkisiä raja-arvoja.

Jos `Array`-olion raja-arvoja ei toisaalta voida muuttaa sijoituksen aikana, silloin arvot ovat kiinteät määrityksen hetkellä, ja asiakkaan olisi mahdollista (vaikkakin raskasta) pitää kirjaa näistä arvoista. Tässä tapauksessa, vaikka olisi mukavaa, että funktiot palauttaisivat nykyiset raja-arvot, tällaiset funktiot eivät olisi osa todellista minimaalista rajapintaa.

Kun oletetaan, että sijoitus voi muuttaa olion raja-arvot, menettelytapa on, että raja-arvojen funktiot voitaisiin esitellä näin:

```
int lowBound() const;
int highBound() const;
```

Koska nämä funktiot eivät muuta oliota, jonka kohdalla niitä pyydetään avuksi, ja koska käytät mieluummin `const`-tyyppistä muuttujaa aina kun voit (katso Kohta 21), nämä molemmat esitellään `const`-tyyppisinä jäsenfunktioina. Edellä mainittu silmukka kirjoitetaan näillä funktioilla seuraavasti:

```
for (int i = a.lowBound(); i <= a.highBound(); ++i)
  cout << "a[" << i << "] = " << a[i] << '\n';
```

On tarpeetonta sanoa, että jotta tällainen silmukka toimisi oliotaulukoille, jotka ovat tyyppiä `T`, `operator<<`-funktio täytyy olla määritettynä olioille, jotka ovat tyyppiä `T`. (Tämä ei ole aivan totta. `T`:lle tai jollekin muulle tyypille, joksi `T` voidaan implisiittisesti konvertoida, täytyy olla olemassa `operator<<`-funktio. Sait kuitenkin varmaan hyvän kuvan.)

Eräät suunnittelijat kiistelisivät varmaan siitä, pitäisikö `Array`-luokan myös sisältää funktio, joka palauttaa elementtien määrän `Array`-oliossa. Elementtien määrä on yksinkertaisesti arvoltaan `highBound()-lowBound()+1`, joten tällainen funktio ei todella ole välttämätön, mutta kerrasta poikki -virheiden toistumisen näkymässä ei olisi kovin huono ajatus lisätä tällainen funktio.

Muita funktioita, jotka voivat osoittautua kannattaviksi tälle luokalle, ovat syötön ja tulostuksen funktiot, kuten myös eräät suhteelliset operaattorit (eli `<`, `>`, `==` jne). Näistä funktioista mikään ei kuitenkaan ole osa minimaalista rajapintaa, koska ne voidaan kaikki toteuttaa silmukan termeillä sisältäen kutsuja `operator[]`-funktioon.

Kohdassa 19 keskustellaan siitä, miksi funktiot kuten `operator<<` ja `operator>>` ja suhteelliset operaattorit toteutetaan säännöllisesti ei-jäsen-ystäväfunktioina jäsenfunktioiden sijasta. Kun näin on asia, älä unohda, että ystäväfunktiot ovat käytännöllisistä syistä osa luokan rajapintaa. Tämä tarkoittaa sitä, että ystäväfunktiot ovat tärkeitä luokan rajapinnan täydellisyydelle ja minimaalisuudelle.

Kohta 19: Eriytä jäsenfunktiot, ei-jäsenfunktiot ja ystäväfunktiot.

Suurin ero jäsenfunktioiden ja ei-jäsenfunktioiden välillä on se, että jäsenfunktiot voivat olla virtuaalisia, mutta ei-jäsenfunktiot eivät. Tästä on tuloksena se, että jos sinulla on funktio, joka pitää sitoa dynaamisesti (katso Kohta 38), sinun täytyy käyttää virtuaalifunktiota, ja tämän virtuaalifunktion täytyy olla jonkin luokan jäsen. Se on näin yksinkertaista. Jos funktiosi ei kuitenkaan tarvitse olla virtuaalinen, vesi alkaa vähän samentua.

Tutki luokkaa, jolla esitetään suhdelukuja:

```
class Rational {
public:
  Rational(int numerator = 0, int denominator = 1);
  int numerator() const;
  int denominator() const;

private:
  ...
};
```

Tämä on aika hyödytön luokka tällaisenaan. (Kun käytetään Kohdan 18 termejä, rajapinta on varmasti minimaalinen, mutta se on *kaukana* valmiista.) Tiedät, että haluaisit tukea aritmeettisia operaattoreita, kuten yhteenlasku, vähennys, kertolasku ja niin edelleen, mutta et ole varma, pitäisikö sinun toteuttaa ne jäsenfunktion, ei-jäsenfunktion vai mahdollisesti jäsenfunktion, jolla on ystävä, avulla.

Kun olet epävarma, ole oliosuuntautunut. Tiedät, että esimerkiksi suhdelukujen kertolasku on sukua `Rational`-luokalle, joten yritä niputtaa toiminto luokkaan tekemällä siitä jäsenfunktio:

```
class Rational {
public:

  ...

  const Rational operator*(const Rational& rhs) const;
};
```

(Jos olet epävarma siitä, miksi tämä funktio on esitelty tällä tavalla - palauttamalla `const`-määre arvonsa tuloksena, mutta ottamalla viittauksen `const`-tyyppiseen muuttujaan argumenttina - tutki Kohtia 21 - 23.)

Nyt voit helpotuksen vallassa kertoa suhdelukuja:

```
Rational oneEighth(1, 8);
Rational oneHalf(1, 2);

Rational result = oneHalf * oneEighth;   // toimii

result = result * oneEighth;             // toimii
```

Mutta et ole tyytyväinen. Haluaisit myös tukea sekatilan toimintoja, jossa `Rational`-luokkia voidaan kertoa esimerkiksi `int`-tyyppisillä muuttujilla. Kun yrität tehdä näin, huomaat kuitenkin, että se toimii vain puolinaisesti:

```
result = oneHalf * 2;                    // toimii
result = 2 * oneHalf;                    // virhe!
```

Tämä on paha enne. Muistathan, että kertolaskun pitäisi olla käänteistä?

Ongelman lähde tulee ilmeiseksi, kun kirjoitat kaksi viimeistä esimerkkiä uudestaan vastaavassa toiminnallisessa muodossaan:

```
result = oneHalf.operator*(2);          // toimii
result = 2.operator*(oneHalf);          // virhe!
```

`oneHalf`-olio on ilmentymä luokasta, joka sisältää `operator*`-funktion, joten kääntäjäsi kutsuvat tuota funktiota. Kokonaisluvulla 2 ei ole kuitenkaan siihen liittyvää luokkaa, ja täten sillä ei ole `operator*`-jäsenfunktiota. Kääntäjäsi etsivät myös ei-jäsen-`operator*`-funktiota (eli sitä, joka on näkyvässä nimiavaruudessa tai on globaali), jota voidaan kutsua tähän tyyliin:

```
result = operator*(2, oneHalf);         // virhe!
```

Ei kuitenkaan ole olemassa ei-jäsen-`operator*`-operaattoria, joka vastaanottaa `int`-arvon ja `Rational`-olion, joten haku epäonnistuu.

Katso uudelleen onnistuvaa kutsua. Näet, että vaikka sen toinen parametri on kokonaisluku 2, `Rational::operator*`-jäsenfunktio ottaa silti `Rational`-olion toisena argumenttinaan. Mitä on tekeillä? Miksi 2 toimii toisessa paikassa mutta ei toisessa?

Tekeillä on implisiittisen tyypin muunnos. Kääntäjäsi tietävät, että välität `int`-tyyppisen arvon ja funktio vaatii `Rational`-olion, mutta ne tietävät myös, että ne voivat loihtia esiin sopivan `Rational`-olion kutsumalla `Rational`-muodostinfunktiota `int`-tyyppisellä arvolla, jonka välitit, joten ne tekevät niin. Toisin sanoen ne kohtelevat kutsua niin kuin olisi kirjoitettu enemmän tai vähemmän tähän tyyliin:

```
const Rational temp(2);          // luo tilapäinen
                                 // Rational-olio 2:sta

result = oneHalf * temp;         // sama kuin
                                 // oneHalf.operator*
                                 // (tilapäinen);
```

Ne tekevät näin tietysti vain silloin, kun ei-`explicit`-tyyppiset muodostinfunktiot ovat mukana kuvassa, koska `explicit`-tyyppisiä muodostinfunktioita ei voi käyttää implisiittisiin muunnoksiin; sitähän `explicit` tarkoittaa. Jos `Rational`-luokka olisi määritetty näin:

```
class Rational {
public:
  explicit Rational(int numerator = 0,     // tämä ctor on
                    int denominator = 1);  // nyt explicit
  ...
  const Rational operator*(const Rational& rhs) const;
  ...
};
```

kumpikaan näistä lauseista ei kääntyisi:

```
result = oneHalf * 2;                    // error!
result = 2 * oneHalf;                    // error!
```

Tämä tuskin kelpaisi tueksi sekatilan aritmetiikkaan, mutta kahden lauseen käyttäytyminen olisi ainakin yhdenmukaista.

Tutkimamme `Rational`-luokka on kuitenkin suunniteltu vastaanottamaan implisiittiset muunnokset sisäänrakennetuista tyypeistä `Rational`-olioiksi - tämän takia `Rational`-olion muodostinfunktiota ei ole esitelty `explicit`-tyyppiseksi. Kun asia on näin, kääntäjät *suorittavat* implisiittisen muunnoksen, joka on pakollinen sallittaessa `result`-muuttujan ensimmäisen sijoituksen kääntyminen. Keikarimaiset kääntäjäsi suorittavat itse asiassa tämänkaltaisen implisiittisen tyyppikonversion tarvittaessa *jokaisen* funktiokutsun *jokaiselle* parametrille. Mutta ne tekevät niin vain parametreille, jotka on *lueteltu parametrilistassa*, ei *koskaan* sille oliolle, jonka kohdalla funktiota pyydetään, eli oliolle, joka vastaa `*this`-osoittimeen jäsenfunktion sisällä. Tästä syystä tämä kutsu toimii:

```
result = oneHalf.operator*(2); // muuntaa int -> Rational
```

ja tämä ei toimi:

```
result = 2.operator*(oneHalf); // ei muunna
                               // int -> Rational
```

Ensimmäinen tapaus sisältää parametrin, joka on lueteltu funktion esittelyssä, mutta toinen tapaus ei sisällä parametriä.

Kaikesta huolimatta haluaisit lisätä tuen sekatilan aritmetiikalle, ja tekotapa on nyt ehkä selvä: tee `operator*`-funktiosta ei-jäsen-funktio, tämä sallii kääntäjien suorittaa implisiittisiä tyyppimuunnoksia *kaikille* argumenteille:

```
class Rational {
  ...                          // ei sisällä operator*
};
```

```
// esittele tämä globaalisena tai nimiavaruuden sisällä
const Rational operator*( const Rational& lhs,
                          const Rational& rhs)
{
  return Rational(lhs.numerator() * rhs.numerator(),
                  lhs.denominator() * rhs.denominator());
}

Rational oneFourth(1, 4);
Rational result;

result = oneFourth * 2;        // hienoa
result = 2 * oneFourth;        // hurraa, se toimii!
```

Tämä on varmasti onnellinen loppu tarinaan, mutta kalvava huoli on silti olemassa. Pitäisikö `operator*`-funktiosta tehdä `Rational`-olion ystävä?

Tässä tapauksessa vastaus on ei, koska `operator*`-funktio voidaan toteuttaa täysin luokan julkisen rajapinnan ehdoilla. Yksi tekotapa nähdään edellä mainitussa lähdetekstissä. Sinun kannattaa välttää ystäväfunktioita aina kun voit, koska kuten tosielämässä, ystävistä on enemmän haittaa kuin hyötyä.

On kuitenkin yleistä funktioille, jotka eivät vielä ole jäseniä, vaikka ne ovatkin käsitteellisesti osa luokan rajapintaa, että ne tarvitsevat pääsyn luokan ei-yleisiin jäseniin.

Esimerkkinä vetäytykäämme taaksepäin tämän kirjan työmyyrällä, `String`-luokalla. Jos yrität kuormittaa `operator>>`- ja `operator<<`-funktioita `String`-olioiden lukemista ja kirjoittamista varten, huomaat pian, että niiden ei pitäisi olla jäsenfunktioita. Jos ne olisivat, sinun olisi täytynyt sijoittaa `String`-olio vasemmalle, kun kutsuit funktioita:

```
// luokka, joka väärin esittelee operator>>:n ja
// operator<<:n jäsenfunktioina
class String {
public:
  String(const char *value);

  ...

  istream& operator>>(istream& input);
  ostream& operator<<(ostream& output);

private:
  char *data;
};

String s;

s >> cin;                      // laillinen, mutta vastoin
                               // sopimusta

s << cout;                     // samat sanat
```

Tämä hämmentäisi kaikkia. Tästä on tuloksena, että näiden funktioiden ei pitäisi olla jäsenfunktioita. Huomaa, että tämä on erilainen tapaus kuin edellä mainittu. Tässä on tavoitteena luonnollinen kutsusyntaksi; olimme aikaisemmin huolestuneita implisiittisistä tyyppimuunnoksista.

Jos olisit suunnittelemassa näitä funktioita, tulisit tämän tyyliseen tulokseen:

```
istream& operator>>(istream& input, String& string)
{
  delete [] string.data;
  lue syöttötiedosta muistiin, ja laita string.data
  osoittamaan siihen
  palauta syöttötieto;
}
ostream& operator<<(ostream& output,
                    const String& string)
{
  return output << string.data;
}
```

Huomaa, että molemmat funktiot tarvitsevat pääsyn `String`-luokan `data`-kenttään, joka on yksityinen. Tiedät jo kuitenkin, että sinun täytyy tehdä näistä funktioista ei-jäsen-tyyppisiä. Sinut on nyrkkeilty nurkkaan eikä sinulla ole vaihtoehtoja: ei-jäsen-funktioista, joilla on pääsy luokan ei-yleisiin jäseniin, täytyy tehdä tuon luokan ystäviä.

Tämän Kohdan harjoituksista on yhteenveto alla, ja niissä oletetaan, että `f` on funktio, jonka yrität esitellä kunnolla ja `C` on luokka, johon se käsitteellisesti liittyy:

- **Virtuaalifunktioiden täytyy olla jäseniä**. Jos `f`:n pitää olla virtuaalinen, tee siitä `C`:n jäsenfunktio.
- **`operator>>`- ja `operator<<`-funktiot eivät koskaan ole jäseniä**. Jos `f` on `operator>>`- tai `operator<<`-funktio, tee `f`:stä ei-jäsen-funktio. Jos `f` tarvitsee lisäksi pääsyn `C`-kielen yksityisiin jäseniin, tee `f`-funktiosta `C`:n ystävä.
- **Vain ei-jäsenfunktiot saavat tyyppimuunnokset vasemmanpuoleisena argumenttina**. Jos `f` tarvitsee tyyppimuunnokset äärimmäisenä vasemmanpuoleisena argumenttina, tee `f`-funktiosta ei-jäsenfunktio. Jos `f` tarvitsee lisäksi pääsyn C:n ei-julkisiin jäseniin, tee `f`-funktiosta `C`:n ystävä.
- **Kaikkien muiden pitäisi olla jäsenfunktioita**. Jos mikään muista tapauksista ei päde, tee `f`:stä `C`:n jäsenfunktio.

Kohta 20: Vältä tietojäseniä julkisessa rajapinnassa.

Tarkastelkaamme asiaa ensin johdonmukaisuuden näkökulmasta. Jos kaikki yleisessä rajapinnassa olevat ovat funktioita, luokkasi asiakkaiden ei tarvitse raapia päätään yrittäen muistaa, käytetäänkö kaarisulkeita silloin, kun ne haluavat pääsyn luokkasi jäseneen. Ne vain *tekevät* niin, koska kaikki ovat funktioita. Tämä voi säästää koko elämänkaaren aikana paljolta pään raapimiselta.

Eikö johdonmukaisuuden argumentti kelpaa sinulle? Miltä kuulostaa se, että funktioiden käyttäminen antaa sinulle tarkemman hallinnan tietojäsenien tavoitettavuuteen? Jos teet tietojäsenestä yleisen, kaikilla on luku- ja kirjoitusoikeus siihen, mutta jos käytät funktioita sen arvon saamiseksi ja määrittämiseksi, voit toteuttaa pääsyn eston, lukuoikeuden sekä luku- ja kirjoitusoikeuden. Hitto, voit jopa toteuttaa kirjoitusoikeudellisen pääsyn, jos haluat:

```
class AccessLevels {
public:
  int getReadOnly() const{ return readOnly; }

  void setReadWrite(int value) { readWrite = value; }
  int getReadWrite() const { return readWrite; }

  void setWriteOnly(int value) { writeOnly = value; }

private:
  int noAccess;                 // ei pääsyä tähän int-
                                // tyyppiin

  int readOnly;                 // vain lukuoikeus tähän
                                // int-tyyppiin

  int readWrite;                // luku-/kirjoitusoikeus
                                // tähän int-tyyppiin

  int writeOnly;                // vain kirjoitusoikeus
                                // tähän int-tyyppiin
};
```

Etkö ole vieläkään vakuuttunut? Sitten on aika ottaa esille varsinaiset aseet: funktionaalinen abstraktio. Jos toteutat pääsyn tietojäseneen funktion kautta, voit korvata tietojäsenen myöhemmin laskennalla ja kukaan luokkaasi käyttävä ei ole yhtään viisaampi.

Oletetaan esimerkiksi, että kirjoitat ohjelmaa, jossa jokin automatisoitu koneisto valvoo ohikulkevien autojen nopeutta. Kun auto menee ohitse, sen nopeus lasketaan ja arvo lisätään kokoelmaan, johon nopeuksien tiedot on kerätty siihen mennessä:

```
class SpeedDataCollection {
public:
  void addValue(int speed);     // lisää uusi tietoarvo
```

```
    double averageSoFar() const; // palauta keskinopeus
};
```

Tutki seuraavaksi jäsenfunktion `averageSoFar` toteutusta. Yksi tapa toteuttaa se on sisällyttää luokkaan jäsenfunktio, joka on juokseva keskiarvo kaikesta nopeustiedosta, joka on siihen mennessä kerätty. Aina kun `averageSoFar`-funktiota kutsutaan, se palauttaa pelkästään tietojäsenen arvon. Erilainen työtapa on, että `averageSoFar` laskee arvonsa uudelleen joka kerta kutsuttaessa, eli se voisi tehdä tämän tutkimalla jokaisen tiedon arvon kokoelmassa.

Ensimmäinen työtapa - juoksevan keskiarvon ylläpitäminen - tekee jokaisesta `SpeedDataCollection`-oliosta isomman, koska muistista täytyy varata tilaa tietojäsenelle, joka tallentaa juoksevan keskiarvon. `averageSoFar`-funktio voidaan kuitenkin toteuttaa erittäin tehokkaasti; se on vain avoin funktio (katso Kohta 33), joka palauttaa tietojäsenen arvon. Käänteisesti ottaen, keskiarvon laskeminen aina pyydettäessä tekee `averageSoFar`-funktion suorituksesta hitaamman, mutta jokainen `SpeedDataCollection`-olio tulee olemaan pienempi.

Kuka sanoo, mikä on paras? Koneessa, jossa muisti on kortilla, ja ohjelmassa, jossa keskiarvoja tarvitaan vain harvoin, keskiarvon laskeminen joka kerta on parempi ratkaisu. Ohjelmassa, jossa keskiarvoja tarvitaan säännöllisesti, nopeus on olennainen osa ja muisti ei ole ongelma, juoksevan keskiarvon käyttö on suositeltavaa. Tärkeä seikka on se, että kun keskiarvoon päästään käsiksi jäsenfunktion avulla, voit käyttää jompaakumpaa toteutustapaa. Tämä on arvokas joustavuuden lähde, jota sinulla ei olisi, jos olisit päättänyt niin, että sisällytät juoksevan keskiarvon tietojäsenen yleiseen rajapintaan.

Tästä on lopputuloksena, että kerjäät hankaluuksia sijoittamalla tietojäsenet yleiseen rajapintaan, joten pelaa varman päälle ja piilota kaikki tietojäsenesi funktionaalisen abstraktion muurin taakse. Jos teet sen *nyt*, heitämme kehään johdonmukaisuuden ja hienosyisen pääsyn hallinnan ilman ylimääräisiä kustannuksia.

Kohta 21: Käytä `const`-määrettä aina kun mahdollista.

`const`-määreessä on se hieno juttu, että se sallii tiettyjen semanttisten rajoitusten määrittämisen - määrättyä oliota *ei* pitäisi muuttaa - ja kääntäjät pakottavat tuon rajoituksen. Se sallii sinun kommunikoida sekä kääntäjien että muiden ohjelmoijien kanssa niin, että arvon tulisi säilyä vakiona. Aina kun tämä on totta, sinun pitäisi varmistaa, että kerrot sen eksplisiittisesti, koska tällä tavalla värväät kääntäjäsi avun varmistamalla, että rajoitusta ei häväistä.

Avainsana `const` on huomattavan monipuolinen. Voit käyttää sitä luokkien ulkopuolella globaaleille tai nimiavaruuden vakioille (katso Kohdat 1 ja 47) ja staattisille olioille (paikallinen tiedostolle tai lohkolle). Sitä voidaan käyttää luokkien sisällä sekä staattisille että epästaattisille tietojäsenille (katso myös Kohta 12).

Voit määrittää osoittimien kohdalla, onko itse osoitin `const`-tyyppinen, onko tieto johon se osoittaa `const`-tyyppinen, molemmat tai ei kumpikaan:

```
char *p              = "Hello";     // ei-const osoitin,
                                    // ei-const tieto†
const char *p        = "Hello";     // ei-const osoitin,
                                    // const-tieto
char * const p       = "Hello";     // const osoitin,
                                    // ei-const tieto
const char * const p = "Hello";     // const osoitin,
                                    // const tieto
```

Tämä kielioppi ei ole niin oikukas kuin miltä näyttää. Vedät mielessäsi pystysuuntaisen viivan osoittimen esittelyn tähtimerkin läpi, ja jos sana `const` on viivan vasemmalla puolella, se mihin *osoitetaan,* on vakio; jos sana `const` on viivan oikealla puolella, *itse osoitin* on vakio; jos `const` on viivan molemmilla puolilla, molemmat ovat vakioita.

Silloin kun osoitettava kohde on vakio, eräät ohjelmoijat luettelevat `const`-määreen ennen tyypin nimeä. Muut luettelevat sen tyypin nimen jälkeen, mutta ennen tähtimerkkiä. Tästä on tuloksena, että seuraavat funktiot vastaanottavat saman parametrityypin:

```
class Widget { ... };
void f1(const Widget *pw);     // f1 ottaa osoittimen
                               // vakio Widget-olioon
void f2(Widget const *pw);     // niin tekee myös f2
```

Koska molemmat muodot ovat olemassa todellisessa lähdetekstissä, sinun kannattaa tottua niihin molempiin.

Eräät `const`-avainsanan tehokkaimmat käyttötavat juontavat alkunsa sen soveltamisesta funktioiden esittelyissä. `const`-määre voi viitata funktion esittelyn sisällä funktion paluuarvoon, yksittäisiin parametreihin ja jäsenfunktioille funktioon sinänsä.

Silloin kun funktio palauttaa vakioarvon, se mahdollistaa asiakkaiden virheiden esiintymistiheyden vähenemisen luopumatta turvallisuudesta tai tehokkuudesta. Itse asiassa, kuten Kohdassa 29 esitetään, `const`-määreen käyttö paluuarvolla mah-

† `"Hello"`-merkkijonon tyyppi on C++-kielen standardin mukaisesti `const char[]`, eli tyyppi, jota melkein aina kohdellaan `const char*` -tyyppisenä. Olisimme täten odottaneet, että olisi `const`-oikeellisuuden rikkomus alustaa `char*`-pohjainen muuttuja merkkijonoliteraalilla kuten `"Hello"`. Käytäntö on kuitenkin niin yleinen C-kielessä, että standardi myöntää erikoisvapauden tämän kaltaisille alustuksille. Niitä kannattaa kuitenkin välttää, sillä niitä kehotetaan välttämään.

dollistaa funktion turvallisuuden ja tehokkuuden *kehittämisen*, joka muuten saattaisi olla ongelmallista.

Tutki esimerkiksi `operator*`-funktion esittelyä Kohdassa 19 esitellyille suhdeluvuille:

```
const Rational operator*(const Rational& lhs,
                         const Rational& rhs);
```

Monet ohjelmoijat katsovat kieroon, kun he näkevät tämän ensimmäisen kerran. Miksi `operator*`-funktion tuloksena pitäisi olla `const`-tyyppinen olio? Siksi, että jos se ei olisi, asiakkaat pystyisivät suorittamaan tämän kaltaisia hirmutekoja:

```
Rational a, b, c;

...

(a * b) = c;                    // sijoita a*b:n
                                // tuotteeseen!
```

En tiedä, miksi kukaan ohjelmoija haluaisi tehdä sijoituksen kahdesta numerosta koostuvaan tuotteeseen, mutta tiedän tämän: olisi suorastaan laitonta, jos `a`, `b` ja `c` olisivat sisäänrakennettua tyyppiä. Yksi käyttäjän määrittelemien hyvien tyyppien tunnusmerkki on se, että ne välttävät vastikkeetonta käyttäytymiseen liittyvää yhteensopimattomuutta sisäänrakennettujen tyyppien kanssa, ja sijoitusten salliminen kahdesta numerosta koostuvaan tuotteeseen tuntuu minusta aika vastikkeettomalta. Kun `operator*`-funktion paluuarvo esitellään tyypillä `const`, estetään tämä, ja tämän takia se on ainoa oikea asia.

`const`-tyyppisissä parametreissä ei ole mitään erityisen uutta - ne toimivat samalla tavalla kuin paikalliset `const`-tyyppiset oliot. `const`-tyyppiset jäsenfunktiot ovat kuitenkin eri juttu.

`const`-tyyppisten jäsenfunktioiden tarkoitus on tietenkin määrittää, mitä jäsenfunktioita pyydetään avuksi `const`-tyyppisille olioille. Monet ihmiset jättävät huomioimatta sen tosiasian, että jäsenfunktiot, jotka eroavat vain niiden `const`-käsittelyssä, voidaan kuitenkin kuormittaa, ja tämä on C++-kielen tärkeä ominaisuus. Tutki `String`-luokkaa jälleen kerran:

```
class String {
public:

  ...

  // operator[]-funktio ei-const-tyyppisille olioille
  char& operator[](int position)
  { return data[position]; }
```

```
  // operator[]-funktio const-tyyppisille olioille
  const char& operator[](int position) const
  { return data[position]; }

private:
  char *data;
};

String s1 = "Hello";
cout << s1[0];                  // kutsuu non-const
                                // String::operator[]
const String s2 = "World";
cout << s2[0];                  // kutsuu const
                                // String::operator[]
```

Kun kuormitat `operator[]`-funktion ja annat eri versioille eri paluuarvot, `const`- ja ei-`const`-tyyppiset `String`-oliot kyetään käsittelemään eri tavalla:

```
String s = "Hello";             // ei-const String-olio

cout << s[0];                   // toimii – luetaan
                                // ei-const String

s[0] = 'x';                     // toimii – kirjoitetaan
                                // ei-const-tyypp. String

const String cs = "World";      // const String-olio

cout << cs[0];                  // toimii – luetaan
                                // const-tyyppinen String

cs[0] = 'x';                    // virhe! – kirjoitetaan
                                // const-tyyppinen String
```

Huomaa muuten, että tässä olevalla virheellä ei ole mitään tekemistä kutsutun `operator[]`-funktion *paluuarvolla*; funktioon `operator[]` tapahtuvat kutsut ovat oikein. Virhe saa alkunsa yrityksestä tehdä sijoitus `const`-tyyppiseen `char&`-muuttujaan, koska se on `operator[]`-funktion `const`-version paluuarvo.

Huomaa myös, että ei-`const`-tyyppisen `operator[]`-funktion paluuarvon täytyy olla viittaus `char`-muuttujaan - `char` itse ei kelpaa. Jos `operator[]`-funktio palauttaisi yksinkertaisen `char`-muuttujan, tämän tyyliset lauseet eivät kääntyisi:

```
s[0] = 'x';
```

Tämä johtuu siitä, että ei ole koskaan laitonta muuttaa sellaisen funktion paluuarvoa, joka palauttaa sisäänrakennetun tyypin. Vaikka se olisi sallittua, tosiasia, että C++-kieli palauttaa oliot arvonsa mukaisesti (katso Kohta 22) tarkoittaisi sitä, että *kopio* `s.data[0]`-muuttujasta muutettaisiin, ei itse `s.data[0]`-muuttujaa. Tämä ei kuitenkaan ole haluamaasi käytöstä.

Otetaan pieni aikalisä filosofian puolelle. Mitä tarkasti ottaen tarkoittaa se, että jäsenfunktio on arvoltaan `const`-tyyppinen? On kaksi vallitsevaa käsitettä: bittitason `const`-käsittely ja käsitteellinen `const`-käsittely.

Bittitason `const`-leiri uskoo, että jäsenfunktio on `const`-tyyppinen, jos ja vain jos se ei muuta mitään olion tietojäsentä (poissulkien ne, jotka ovat staattiset), toisin sanoen, jos se ei muuta yhtään olion sisällä olevaa bittiä. Bittitason `const`-käsittelyssä on se mukava asia, että väärinkäytökset on helppo huomata: kääntäjät etsivät vain tietojäseniin kohdistuvia sijoituksia. Bittitason `const`-käsittely on itse asiassa C++-kielen määritys `const`-käsittelystä, ja `const`-tyyppisen jäsenfunktion ei sallita muuttaa yhtään avuksi pyydetyn olion tietojäsenistä.

Valitettavasti monet jäsenfunktiot, jotka eivät toimi kovin `const`-tyyppisesti, läpäisevät bittitason testin. Varsinkin silloin, kun jäsenfunktio, joka muuttaa sen, mihin osoitin *osoittaa*, ei säännöllisesti toimi `const`-tyyppisesti. Mutta jos vain osoitin on oliossa, funktio on bittitasolla `const`-tyyppinen, ja kääntäjät eivät valita. Tämä voi johtaa vastanäkemykselliseen käyttäytymiseen:

```
class String {
public:
  // muodostinfunktio saa tiedon osoittamaan kopioon
  // siitä, mihin arvo osoittaa
  String(const char *value);

  ...

  operator char *() const { return data;}

private:
  char *data;
};

const String s = "Hello";      // esittele vakio-olio

char *nasty = s;               // kutsuu op char*() const

*nasty = 'M';                  // muuttaa s.data[0]

cout << s;                     // kirjoittaa "Mello"
```

Jotain on varmasti väärin silloin, kun luot vakio-olion määrätyllä arvolla ja pyydät sille avuksi pelkästään `const`-tyyppisiä jäsenfunktioita, ja pystyt silti muuttamaan sen arvoa! (Jos haluat lukea yksityiskohtaisemmin keskustelun esimerkistä, katso Kohta 29.)

Tämä johtaa `const`-tyyppisyyden käsitteeseen. Tähän filosofiaan takertuneet inttävät, että `const`-tyyppinen jäsenfunktio voi muuttaa sen olion eräitä bittejä, joiden kohdalla sitä pyydetään avuksi, mutta vain sillä tavoin, etteivät sen asiakkaat voi saada selville. `String`-luokkasi voisi esimerkiksi aina pyydettäessä haluta viedä olion pituuden cache-muistiin:

```
class String {
public:
  // muodostin saa tiedon osoittamaan kopioon
  // siitä, mihin arvo osoittaa
  String(const char *value): lengthIsValid(false) { ... }

  ...

  size_t length() const;

private:
  char *data;

  size_t dataLength;            // merkkijonon viimeinen
                                // laskettu pituus

  bool lengthIsValid;           // onko pituus tällä
                                // hetkellä oikea
};

size_t String::length() const
{
  if (!lengthIsValid) {
    dataLength = strlen(data);  // virhe!
    lengthIsValid = true;       // virhe!
  }

  return dataLength;
}
```

Tämä `length`-muuttujan toteutustapa ei varmasti ole bittitasolla `const`-tyyppinen - sekä `dataLength`- että `lengthIsValid`-muuttujia voidaan muuttaa - vaikka tuntuukin siltä, että tämä on hyvin perusteltua `const`-tyyppisille `String`-olioille. Tulet huomaamaan, että kääntäjät ovat kunnioittaen eri mieltä; ne vaativat bittitason `const`-tyyppisyyden. Mitä teet?

Ratkaisu on yksinkertainen: ota hyöty siitä `const`-määreelle liittyvästä heilumistilasta, jonka C++-kielen standardisointikomitea ajattelevaisesti sisällytti juuri tämän tyyppisiin tilanteisiin. Tämä heilumistila ottaa `mutable`-avainsanan muodon. Silloin, kun sitä sovelletaan epästaattisille tietojäsenille, `mutable` vapauttaa nuo jäsenet bittitason `const`-tyyppisyyden rajoituksista:

```
class String {
public:

  ...                           // sama kuin edellä

private:
  char *data;
```

```
  mutable size_t dataLength;          // nämä tietojäsenet ovat
                                      // nyt muutettavissa; ne
  mutable bool lengthIsValid;         // voidaan muuttaa
                                      // missä tahansa, jopa
                                      // const-tyyppisten
};                                    // jäsenfunkt. sisällä

size_t String::length() const
{
  if (!lengthIsValid) {
    dataLength = strlen(data);  // nyt toimii
    lengthIsValid = true;       // toimii myös
  }

  return dataLength;
}
```

`mutable` on hieno ratkaisu "bittitason const-tyyppisyys ei ole aivan sitä, mitä minulla oli mielessäni" -ongelmaan, mutta se lisättiin suhteellisen myöhään C++-kieleen standardisointivaiheessa, joten kääntäjäsi eivät ehkä vielä tue sitä. Jos näin on, sinun täytyy alentua C++-kielen pimeisiin kätköihin, missä elämä on halpaa ja `const`-tyyppisyys voidaan saada aikaan tekemällä muunnos.

`this`-osoitin käyttäytyy C-luokan jäsenfunktion sisällä aivan kuin se olisi määritetty seuraavasti::

```
C * const this;               // ei-const-tyyppisille
                              // jäsenfunktioille

const C * const this;         // const-tyyppisille
                              // jäsenfunktioille
```

Tässä tapauksessa sinun täytyy vain saada ongelmallinen versio `String::length`-funktiosta (eli se, jonka voisit korjata `mutable`-muuttujalla, jos kääntäjäsi tukisivat sitä) toimimaan sekä `const`- että ei-`const`-olioille muuttamalla `this`-muuttujan tyypin arvosta `const C*` arvoon `const C* const`. Et voi tehdä sitä suoraan, mutta voit väärentää sen alustamalla paikallisen osoittimen osoittamaan samaan olioon kuin mihin `this` osoittaa. Sinulla on tämän jälkeen pääsy jäseniin, joita haluat muuttaa paikallisen osoittimen avulla:

```
size_t String::length() const
{
  // tee paikallinen versio this-muuttujasta, joka
  // ei ole osoitin const-tyyppiseen muuttujaan
  String * const localThis =
    const_cast<String * const>(this);

  if (!lengthIsValid) {
    localThis->dataLength = strlen(data);
    localThis->lengthIsValid = true;
  }
```

```
  return dataLength;
}
```

Tämä ei ole kaunista, mutta joskus ohjelmoijan on tehtävä mitä ohjelmoijan on tehtävä.

Ei ole tietenkään takuita, että se toimii ja joskus vanhassa const-tyyppisyyden muuntaminen -tempussa siitä ei ole takuita. Varsinkin jos se, mihin olio `this` osoittaa, on todella `const`-tyyppinen, eli se on esitelty `const`-tyyppisenä määrityshetkellään, tulokset `const`-tyyppisyyden muuntamisessa ovat määrittelemättömät. Jos haluat muuntaa `const`-tyyppisyyden yhdessä jäsenfunktiossasi, sinun on parasta varmistaa, että olio, jonka kohdalla suoritat muuntamisen, ei ollut alun perin määritetty `const`-tyyppisenä.

On myös toinen hetki, jolloin `const`-tyyppisyyden muuntaminen voi olla sekä käytännöllistä että turvallista. Se on silloin, kun sinulla on `const`-olio, jonka haluat välittää funktiolle vastaanottamalla ei-`const`-tyyppisen parametrin, ja *tiedät, että parametriä ei tulla muuttamaan funktion sisällä*. Toinen ehto on tärkeä, koska on aina turvallista muuntaa sen olion `const`-tyyppisyys, joka tullaan vain lukemaan - ei kirjoittamaan - vaikka tuo olio oli alun perin määritetty `const`-tyyppisenä.

Eräät kirjastot ovat esimerkiksi olleet tunnettuja siitä, että ne esittelevät `strlen`-funktion väärin seuraavasti:

```
size_t strlen(char *s);
```

`strlen`-funktio ei varmastikaan aio muuttaa sitä, mihin `s` osoittaa - ei ainakaan se `strlen`, jonka kanssa minä vartuin. Olisi kuitenkin väärin tästä esittelystä johtuen kutsua niitä osoittimille, jotka ovat tyyppiä `const char*`. Päästäksesi ongelmasta voit turvallisesti muuntaa tällaisten osoittimien `const`-tyyppisyyden silloin, kun välität ne `strlen`-funktiolle:

```
const char *klingonGreeting = "nuqneH";    // "nuqneH" on
                                           // "Hello"
                                           // Klingon-
                                           // kielellä
size_t length =
  strlen(const_cast<char*>(klingonGreeting));
```

Älä kuitenkaan rupea hienokäytöksiseksi. Sen on taattu toimivan vain, jos kutsuttu funktio, tässä tapauksessa `strlen`, ei yritä muuttaa sitä, mihin sen parametri osoittaa.

Kohta 22: Suosi parametrien välittämistä viittausparametreillä arvoparametrien sijasta.

C-kielessä kaikki välitetään arvoparametreinä ja C++-kieli kunnioittaa tätä perintöä omaksumalla arvoparametreillä välittämisen sopimuksen oletusarvona. Jos et määritä

muuta, funktion parametrit alustetaan todellisten argumenttien *kopioilla*, funktioiden kutsujat vastaanottavat funktion palauttaman arvon *kopion*.

Kuten korostin tämän kirjan esipuheessa, tuon olion luokan kopiomuodostin määrittää tarkoituksen, että olio välitetään arvoparametrillä. Tämä voi tehdä arvolla välittämi-sestä äärimmäisen kalliin toiminnan. Tutki esimerkiksi seuraavaa (koko lailla keksittyä) luokkahierarkiaa:

```
class Person {
public:
  Person();                    // parametrit jätetty pois
                               // yksinkertaisuuden vuoksi
  ~Person();

  ...

private:
  string name, address;
};

class Student: public Person {
public:
  Student();                   // parametrit jätetty pois
                               // yksinkertaisuuden vuoksi
  ~Student();

  ...

private:
  string schoolName, schoolAddress;
};
```

Tutki nyt yksinkertaista funktiota `returnStudent`, joka ottaa `Student`-argumentin (arvoparametrillä) ja palauttaa sen välittömästi (myös arvoparametrillä) sekä kutsun tuohon funktioon:

```
Student returnStudent(Student s) { return s; }

Student plato;                 // Platon opiskeli
                               // Sokrateen alaisuudessa

returnStudent(plato);          // kutsu returnStudent
```

Mitä tapahtuu tämän harmittomalta näyttävän funktiokutsun aikana?

Tässä yksinkertainen selitys: `s` alustetaan arvolla `plato` kutsumalla `Student`-luokan kopiomuodostinta. `Student`-luokan kopiomuodostinta kutsutaan sitten uudestaan alustamaan funktion palauttama olio `s`:llä. Seuraavaksi kutsutaan `s`:n tuhoajafunktiota. Lopulta kutsutaan tuhoajafunktiota `returnStudent`-funktion palauttamalle oliolle. Joten tämän mitään tekemättömän funktion kustannukset ovat kaksi kutsua `Student` kopiomuodostimeen ja kaksi kutsua `Student`-tuhoajafunktioon.

Mutta odota, lisää seuraa! `Student`-oliolla on sisällään kaksi `string`-oliota, joten joka kerta, kun muodostat `Student`-olion, sinun täytyy muodostaa myös kaksi `string`-oliota. `Student`-olio periytyy myös `Person`-oliosta, joten joka kerta, kun muodostat `Student`-olion, sinun täytyy muodostaa myös `Person`-olio. `Person`-oliolla on kaksi ylimääräistä `string`-oliota sisällään, joten jokainen `Person`-muodostus perii kaksi `string`-muodostusta lisää. Lopputuloksena on, että `Student`-olion välittäminen arvoparametrillä johtaa yhteen kutsuun `Student`-olion kopiomuodostimeen, yhteen kutsuun `Personin` kopiomuodostimeen, ja neljä kutsua `string`-olion kopiomuodostimeen. Kun `Student`-olion kopio tuhotaan, jokaiseen muodostinkutsuun täytyy täsmätä tuhoajafunktiokutsu, joten `Studentin` välittämisen arvoparametrillä kokonaiskustannukset ovat kuusi muodostinfunktiota ja kuusi tuhoajafunktiota. Koska funktio `returnStudent` käyttää arvoparametrillä välittämistä kaksi kertaa (kerran parametrille, kerran paluuarvolle), tuon funktion kutsun kokonaiskustannukset ovat *kaksitoista* muodostinfunktiota ja *kaksitoista* tuhoajafunktiota!

C++-kielen kääntäjä-kirjoittajien maailmassa tämä on rehellisesti sanoen huonoimmasta päästä oleva skenaario. Kääntäjien sallitaan eliminoida eräitä kopiomuodostimien kutsuja (C++-standardi - katso Kohta 50 - kuvaa ne tarkat olosuhteet, jolloin tämänkaltainen taikuus sallitaan suoritettavaksi). Erääät kääntäjät ottavat hyödyn tästä luvasta optimoida. Ennen kuin näistä optimoinneista sellaisia, että ne ovat kaikkialla läsnä, sinun täytyy kuitenkin olla tietoinen niistä kustannuksista, joita syntyy, kun olioita välitetään arvoparametrillä.

Välttääksesi näitä potentiaalisen ylettömiä kustannuksia, sinun ei pidä välittää asioita arvoparametrillä, vaan viittausparametrillä:

```
const Student& returnStudent(const Student& s)
{ return s; }
```

Tämä on paljon tehokkaampi tapa: muodostinfunktioita tai tuhoajafunktioita ei kutsuta, koska mitään olioita ei luoda.

Parametrien välittäminen viittausparametrillä sisältää toisen edun: sillä vältetään se, mitä joskus kutsutaan "paloitteluongelmaksi". Kun periytetty luokkaolio välitetään kantaluokan olioina, kaikki ne erikoispiirteet, jotka saavat ne käyttäytymään kuten periytetyt luokat, on "paloiteltu" pois, ja jäljellä on yksinkertainen kantaluokka. Tätä tuskin haluat koskaan. Oletetaan esimerkiksi, että työskentelet luokkajoukon parissa, jolla toteutetaan graafinen ikkunointijärjestelmä:

```
class Window {
public:
  string name() const;            // palauttaa ikkunan nimen
  virtual void display() const;   // piirrä ikkuna ja sisält.
};
```

```
class WindowWithScrollBars: public Window {
public:
  virtual void display() const;
};
```

Kaikilla `Window`-olioilla on nimi, johon päästään käsiksi `name`-funktion kautta, ja kaikki ikkunat voidaan näyttää pyytämällä avuksi `display`-funktiota. Se tosiasia, että `display` on virtuaalifunktio, kertoo sinulle, että tapa, jolla yksinkertaiset kantaluokan `Window`-oliot näytetään, on omiaan eroamaan tavasta, jolla hienot, kalliit `WindowWithScrollBars`-oliot näytetään (katso Kohdat 36 ja 37).

Oletetaan nyt, että haluaisit kirjoittaa funktion, joka tulostaa ikkunan nimen ja näyttää sitten ikkunan. Tässä on *väärä* tapa kirjoittaa sellainen funktio:

```
// funktio, joka kärsii paloitteluongelmasta
void printNameAndDisplay(Window w)
{
  cout << w.name();
  w.display();
}
```

Tutki mitä tapahtuu, kun kutsut tätä funktiota `WindowWithScrollBars`-oliolla:

```
WindowWithScrollBars wwsb;

printNameAndDisplay(wwsb);
```

Parametri `w` muodostetaan - muistathan, että se välitetään arvoparametrillä? - *`Window`*-oliona, ja kaikki erikoistieto, joka sai `wwsb`-olion käyttäytymään kuin `WindowWithScrollBars`-olio, paloitellaan pois. w tulee aina toimimaan `printNameAndDisplay`-funktion sisällä kuin `Window`-luokan olio (se *on* `Window`-luokan olio), riippumatta sen olion tyypistä, joka välitetään funktiolle. Varsinkin `printNameAndDisplay`-funktion sisällä oleva `display`-kutsu kutsuu *aina* `Window::display`-funktiota, se ei koskaan kutsu `WindowWithScrollBars::display`-funktiota.

Paloitteluongelma hoidetaan välittämällä `w` viittausparametrinä:

```
// funktio, joka ei kärsi paloitteluongelmasta
void printNameAndDisplay(const Window& w)
{
  cout << w.name();
  w.display();
}
```

`w` toimii nyt riippumatta sille välitetystä ikkunatyypistä. Korostaaksesi sitä, että tämä funktio ei ole muuttanut `w`-parametriä, vaikka se on välitetty viittausparametrinä, olet noudattanut Kohdan 21 ohjetta ja julistanut sen `const`-tyyppiseksi; hyvin toimittu.

Viittausparametrillä välittäminen on hieno asia, mutta se johtaa tiettyihin omiin komplikaatioihin, joista huonomaineisin on peitenimen käyttö, Kohdassa 17 käsitelty aihe. On lisäksi tärkeää tunnistaa se, että asioita *ei* joskus voida välittää viittausparametrillä; katso Kohta 23. Raaka totuus on lopulta se, että viittausparametrit *toteutetaan* lähes aina osoittimina, joten välittäminen viittausparametrillä tarkoittaa usein itse asiassa osoitinparametrillä välittämistä. Tästä on tuloksena, että jos sinulla on pieni olio - esimerkiksi `int`-tyyppinen - voi itse asiassa olla tehokkaampaa välittää se arvoparametrillä kuin viittausparametrillä.

Kohta 23: Älä yritä palauttaa viittausta silloin, kun sinun pitää palauttaa olio.

Sanotaan, että Albert Einstein antoi kerran tämän ohjeen: tee asiat mahdollisimman yksinkertaisiksi, mutta älä kuitenkaan yksinkertaisimmiksi. Sama asia C++-kielessä voisi olla, että tehdään asiat mahdollisimman tehokkaiksi, mutta ei kuitenkaan yhtään tehokkaammiksi.

Silloin kun ohjelmoijat tarttuvat tehokkuussotkuun, jonka arvoparametrillä välittäminen tarjoaa olioille (katso Kohta 22), heistä tulee ristiretkeilijöitä, jotka kitkevät arvoparametrillä välittämisen pahuuden juuret, missä ne sitten ovatkin piilossa. Heltymättömänä tavoitellessaan viittausparametrillä välittämisen puhtautta, he muuttumatta tekevät saman kohtalokkaan virheen: he alkavat välittää viittauksia olioihin, joita ei ole olemassa. Tämä ei ole hyvä asia.

Tutki luokkaa, joka esittää suhdelukuja, mukaan lukien ystäväfunktion (katso Kohta 19), jolla kerrotaan kaksi suhdelukua yhteen:

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

  ...

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

friend
  const Rational                    // katso Kohdasta 21
    operator*(const Rational& lhs,  // miksi paluuarvo on
              const Rational& rhs); // tyypiltään const
};

inline const Rational operator*(const Rational& lhs,
                                const Rational& rhs)
{
  return Rational(lhs.n * rhs.n, lhs.d * rhs.d);
}
```

Tämä versio `operator*`-funktiosta palauttaa selvästi tuloksena olevan olion arvoparametrillä, ja lintsaisit ammattivelvollisuuksistasi, jos et murehtisi tuon olion muodostamisen ja tuhoamisen kustannuksia. Toinen selvä asia on, että olet saita, etkä halua maksaa tällaisesta tilapäisestä oliosta, jos sinun ei tarvitse. Joten kysymys kuuluu: tarvitseeko sinun maksaa?

Sinun ei tarvitse, jos voit palauttaa sen sijaan viittauksen. Mutta muista, että viittaus on vain *nimi* jollekin *olemassa olevalle* oliolle. Aina kun näet viittauksen esittelyn, sinun pitäisi välittömästi kysyä itseltäsi, mille se on toinen nimi, koska sen täytyy olla toinen nimi *jollekin*. Jos funktio palauttaa viittauksen `operator*`-funktion tapauksessa, sen täytyy palauttaa se johonkin muuhun jo olemassaolevaan `Rational`-olioon, joka sisältää kahdesta keskenään kerrottavasta oliosta koostuvan tuotteen.

Ei varmastikaan ole syytä odottaa, että tällainen olio on olemassa ennen kutsua `operator*`-funktioon. Tämä tarkoittaa sitä, että jos sinulla on

```
Rational a(1, 2);              // a = 1/2
Rational b(3, 5);              // b = 3/5
Rational c = a * b;            // c pitäisi olla 3/10
```

tuntuu kohtuuttomalta odottaa, että on jo olemassa suhdeluku, jolla on arvo 3/10. Ei, jos `operator*`-funktion täytyy palauttaa viittaus tällaiseen numeroon, sen täytyy luoda tuo numero-olio itse.

Funktio voi luoda uuden olion vain kahdella tavalla: pinossa tai keossa. Luominen pinoon on saatu aikaan määrittelemällä paikallinen muuttuja. Voisit yrittää kirjoittaa oman `operator*`-funktiosi tätä strategiaa käyttämällä seuraavasti:

```
// ensimmäinen väärä tapa kirjoittaa tämä funktio
inline const Rational& operator*(const Rational& lhs,
                                 const Rational& rhs)
{
  Rational result(lhs.n * rhs.n, lhs.d * rhs.d);
  return result;
}
```

Tämä työtapa voidaan hylätä suoralta kädeltä, koska tavoitteesi oli välttää muodostinfunktion kutsu, ja `result`-muuttuja täytyisi muodostaa samalla tavalla kuin mikä tahansa muukin olio. Tällä funktiolla on lisäksi vakavampi ongelma siinä, että se palauttaa viittauksen paikalliseen olioon. Tämä on virhe, jota käsitellään syvällisesti Kohdassa 31.

Sinulla on mahdollisuus muodostaa olio keossa ja palauttaa viittaus sitten siihen. Kekopohjaiset oliot syntyvät käyttämällä `new`-operaattoria. Voisit kirjoittaa `operator*`-funktion siinä tapauksessa tällä tavalla:

```
// toinen väärä tapa kirjoittaa tämä funktio
inline const Rational& operator*(const Rational& lhs,
                                 const Rational& rhs)
{
  Rational *result =
    new Rational(lhs.n * rhs.n, lhs.d * rhs.d);
  return *result;
}
```

Sinun täytyy *silti* maksaa muodostinkutsusta, koska `new`-operaattorin dynaamisesti varaama muisti alustetaan kutsumalla asianmukaista muodostinfunktiota (katso Kohta 5), mutta nyt sinulla on eri ongelma: kuka soveltaa `delete`-operaattoria olioon, joka loihdittiin esiin käyttämällä `new`-operaattoria?

Kyseessä on itse asiassa takuuvarma muistivuoto. Vaikka `operator*`-funktion kutsujat pystyttäisiin taivuttelemaan vastaanottamaan funktion tuloksen osoite ja käyttämään siinä `delete`-operaattoria (tähtitieteellisen epätodennäköistä - Kohdassa 31 nähdään, miltä lähdetekstin pitäisi näyttää), monimutkaiset ilmaisut saisivat aikaan nimettömiä tilapäisiä muuttujia, joihin ohjelmoijat eivät koskaan pystyisi pääsemään käsiksi. Esimerkiksi molemmat kutsut

```
Rational w, x, y, z;
w = x * y * z;
```

`operator*`-funktioon saisivat aikaan nimettömiä tilapäismuuttujia, joita ohjelmoijat eivät koskaan näkisi, eivätkä täten pystyisi poistamaan. (Katso jälleen Kohta 31.)

Mutta ajattelet varmaan, että olet ovelampi kuin tavallinen Virtanen - tai keskiverto ohjelmoija. Huomaat ehkä, että sekä pino- että keko-tyyppiset työtavat kärsivät siitä, että joutuvat kutsumaan muodostinfunktiota jokaiselle `operator*`-funktion palauttamalle tulokselle. Sinulle palautuu ehkä mieleen, että alkuperäinen tavoitteemme oli välttää tämän kaltaisia muodostinfunktioiden avuksipyytämisiä. Ajattelet varmaan, että tiedät tavan välttää muut paitsi yhden muodostinkutsun. Ehkä sinulle juolahtaa mieleen seuraava toteutustapa, joka perustuu `operator*`-funktioon palauttamalla viittauksen *staattiseen* `Rational`-olioon, eli siihen, joka on määritetty funktion *sisällä*:

```
// kolmas väärä tapa kirjoittaa tämä funktio
inline const Rational& operator*(const Rational& lhs,
                                 const Rational& rhs)
{
  static Rational result;     // staattinen olio johon
                              // palautetaan viittaus

  kerro jotenkin lhs ja rhs keskenään ja sijoita
  tuloksena oleva arvo tuloksen sisälle;

  return result;
}
```

Tämä näyttää lupaavalta, paitsi että kun yrität muodostaa todellista C++-koodia edellämainitulla kursivoidulla pseudokoodilla, huomaat, että on melkein mahdotonta antaa `result`-muuttujalle oikeaa arvoa pyytämättä avuksi `Rational`-muodostinfunktiota, ja tällaisen kutsun välttäminen oli syynä koko peliin. Olettakaamme, että onnistut kuitenkin löytämään tavan, koska mikään älykkyyden määrä ei voi viime kädessä pelastaa tätä huono-onnista työtapaa.

Näet syyn tutkimalla tätä täydellisen järkevää asiakaskoodia:

```
bool operator==(const Rational& lhs,     // operator==
                const Rational& rhs);    // Rational-
                                         // olioille

Rational a, b, c, d;

...

if ((a * b) == (c * d))  {
  tee se mikä on sopivaa, kun tuotteet ovat samanarvoisia;
} else  {
  tee se mikä on sopivaa, kun tuotteet eivät ole samanarv.;
}
```

Aprikoi seuraavaksi tätä: ilmaisu `((a*b) == (c*d))` arvottuu *aina* arvoksi `true`, riippumatta muuttujien `a`, `b`, `c` ja `d` arvoista!

Tämä harmittava käytös on helpointa ymmärtää kirjoittamalla samanarvoisuuden testi uudelleen vastaavassa toiminnallisessa muodossaan:

```
if (operator==(operator*(a, b), operator*(c, d)))
```

Huomaa, että silloin kun `operator==`-funktiota kutsutaan, `operator*`-funktioon on jo olemassa *kaksi* aktiivista kutsua, jotka molemmat palauttavat viittauksen `operator*`-funktion sisällä olevaan staattiseen `Rational`-olioon. `operator*`-funktiota pyydetään täten vertaamaan staattisen `operator*`-funktion sisällä olevan `Rational`-olion arvoa `operator*`-funktion sisällä olevan `Rational`-olion arvoon. Olisi todella yllättävää, jos ne eivät olisi samanarvoisia. Aina.

Onnella tämä riittänee vakuuttamaan sinut siitä, että viittauksen palauttaminen funktiosta kuten `operator*` on ajan tuhlausta, mutta en ole niin naiivi, että uskoisin onnen aina riittävän. Eräät teistä - ja tiedät kyllä ketkä - ajattelevat juuri tällä hetkellä: "Jos *yksi* staattinen ei riitä, niin ehkä staattinen *taulukko* hoitaa asian..."

Lopeta, ole hyvä. Emmekö ole jo kärsineet tarpeeksi?

En pysty jalostamaan tätä työtapaa esimerkkikoodilla, mutta voin kyllä hahmottaa, miksi jopa käsitteellä *viihdyttäminen* pitäisi saada sinut punastumaan häpeästä. Ensinnäkin, sinun täytyy valita *n*, eli taulukon koko. Jos *n* on liian pieni, voi olla, että sinulta ovat loppuneet paikat, jonne paluuarvoja varastoidaan, missä tapauksessa et ole hyö-

tynyt mitään siitä yhden `static`-tyyppisen olion suunnittelusta, jonka juuri saatoimme häpeään. Mutta jos *n* on liian iso, vähennät ohjelmasi suorituskykyä, koska taulukon *jokainen* olio muodostetaan silloin, kun funktiota kutsutaan ensimmäisen kerran. Tämä maksaa sinulle *n*-määrän muodostinfunktioita ja *n*-määrän tuhoajafunktioita, vaikka kyseessä olevaa funktiota kutsuttaisiin vain kerran. Jos "optimointi" on prosessi, jolla parannetaan ohjelman suorituskykyä, tämänkaltaista juttua pitäisi kutsua "pessimoinniksi". Mieti lopuksi, kuinka olisit sijoittanut tarvitsemasi arvot taulukon olioihin ja mitä sen tekeminen olisi maksanut. Suorin tapa siirtää arvo olioiden välillä on sijoittaminen, mutta mikä on sijoittamisen hinta? Yleensä ottaen se on sama kuin kutsu tuhoajafunktioon (vanhan arvon tuhoamiseksi) sekä kutsu muodostinfunktioon (uuden arvon kopiointi). Mutta tavoitteesihan oli välttää muodostamisen ja tuhoamisen kustannukset! Myönnä pois: tämä työtapa ei yksinkertaisesti tule onnistumaan.

Oikea tapa kirjoittaa funktio, jonka täytyy palauttaa uusi olio, on antaa funktion palauttaa uusi olio. Tämä tarkoittaa joko seuraavan lähdetekstin (jonka näimme ensimmäisen kerran sivulla 96) varastointia tai muuta oleellisesti vastaavaa:

```
inline const Rational operator*(const Rational& lhs,
                                const Rational& rhs)
{
  return Rational(lhs.n * rhs.n, lhs.d * rhs.d);
}
```

Sinulle voi varmasti koitua velkaa `operator*`-funktion palautusarvon muodostamisen ja tuhoamisen kustannuksista, mutta ajan mittaan se on aika pieni hinta oikeanlaisesta käyttäytymisestä. Sitä paitsi laskua, joka sinua kauhistuttaa niin kovasti, ei ehkä koskaan tule. Kuten kaikki ohjelmointikielet, C++ sallii kääntäjien toteuttajien ottaa käyttöön eräitä optimointeja, joilla parannetaan luodun koodin suorituskykyä, ja joissakin tapauksissa tulee myös ilmi, että `operator*`-funktion paluuarvo voidaan turvallisesti eliminoida. Kun kääntäjät hyötyvät tuosta tosiasiasta (ja nykyiset kääntäjät usein tekevät niin), ohjelmasi jatkaa käyttäytymällä oletetulla tavalla, se vain tekee sen nopeammin kuin odotit.

Kaikki johtaa tähän: kun päätät, palautatko viittauksen vai olion, tehtäväsi on tehdä valinta, joka tekee oikein. Anna kääntäjien toimittajien painiskella laskelmoidessaan sitä, kuinka valinnasta tehdään mahdollisimman edullinen.

Kohta 24: Valitse huolellisesti funktioiden kuormittamisen ja parametrien oletusarvojen välillä.

Hämmennys funktioiden kuormittamisen ja parametrien oletusarvojen määrittämisen yllä juontaa juurensa siitä, että ne molemmat sallivat yksittäisen funktionimen kutsumisen useammalla kuin yhdellä tavalla:

```
void f();                       // f on kuormitettu
void f(int x);

f();                            // kutsuu f()-funktiota
f(10);                          // kutsuu f(int)-funktiota

void g(int x = 0);              // g:llä on oletus
                                // parametriarvo

g();                            // kutsuu g(0)-funktiota
g(10);                          // kutsuu g(10)-funktiota
```

Kumpaa sitten pitäisi käyttää?

Vastaus riippuu kahdesta muusta kysymyksestä. Ensinnäkin, onko olemassa arvo, jota voit käyttää oletuksena? Toiseksi, kuinka montaa algoritmia haluat käyttää? Jos voit yleisesti ottaen valita järkevän oletusarvon ja haluat ottaa käyttöön vain yhden algoritmin, käytetään oletusparametrejä (katso myös Kohta 38). Muuten käytetään funktioiden kuormittamista.

Tässä on funktio, joka laskee viiden `int`-tyyppisen muuttujan enimmäisarvon. Tämä funktio käyttää - vedä syvään henkeä ja terästäydy - `std::numeric_limits<int>::min()`:iä oletuksena olevana parametrin arvona. Tästä hieman lisää hetken kuluttua, mutta tässä on ensin koodi:

```
int max(int a,
        int b = std::numeric_limits<int>::min(),
        int c = std::numeric_limits<int>::min(),
        int d = std::numeric_limits<int>::min(),
        int e = std::numeric_limits<int>::min())
{
  int temp = a > b ? a : b;
  temp = temp > c ? temp : c;
  temp = temp > d ? temp : d;
  return temp > e ? temp : e;
}
```

Rauhoitu nyt. `std::numeric_limits<int>::min()` on vain hieno uudenaikainen tapa, jolla perus-C++-kirjasto määrittää sen, mitä C-kieli määrittää `INT_MIN`-makron avulla otsikkotiedostossa `<limits.h>`: se on pienin mahdollinen arvo `int`-muuttujalle, oli sitten C++-kielisen lähdetekstisi kääntäjä mikä tahansa. Tämä on tosiaan poikkeus siitä niukkasanaisuudesta, josta C-kieli on tun-

nettu, mutta kaikkien näiden kaksoispisteiden ja lauseopillisen strykniinin takana on metodi.

Oletetaan, että haluaisit kirjoittaa funktiomallin, joka vastaanottaa parametrinään minkä tahansa sisäänrakennetun numeerisen tyypin, ja haluaisit mallin luomien funktioiden tulostavan vähimmäisarvon, joka on esitettävissä alustavalla tyypillään. Mallisi näyttäisi tällaiselta:

```
template<class T>
void printMinimumValue()
{
  cout << pienin T:n esiteltävissä oleva arvo;
}
```

Tämän funktion kirjoittaminen on vaikeaa, sillä sinulla on työkaluina vain otsikkotiedostot `<limits.h>` ja `<float.h>`. Et tiedä mikä `T` on, joten et tiedä, pitääkö sinun tulostaa `INT_MIN` tai `DBL_MIN` vai mitä.

Kun nämä vaikeudet halutaan sivuuttaa, perus-C++-kirjasto (katso Kohta 49) määrittää otsikossa `<limits>` luokkamallin `numeric_limits`, joka itse määrittää useita staattisia jäsenfunktioita. Jokainen funktio palauttaa tietoa tyypistä, joka instantioi mallin. Tämä tarkoittaa sitä, että `numeric_limits<int>`:n funktiot palauttavat tietoa `int`-tyypistä, `numeric_limits<double>`:n funktiot palauttavat tietoa `double`-tyypistä ja niin edelleen. `numeric_limits`:n funktioiden joukossa on funktio `min`. Funktio `min` palauttaa pienimmän esiteltävissä olevan arvon alustavalle tyypille, joten `numeric_limits<int>::min()`-funktio palauttaa pienimmän esiteltävissä olevan kokonaisluvun arvon.

`printMinimumValue`-muuttujan kirjoittaminen on niin helppoa, kuin se vain voi olla funktiolla `numeric_limits` (joka, kuten melkein kaikki peruskirjastossa oleva, sijaitsee `std`-nimiavaruudessa - katso Kohta 28; `numeric_limits` itse on otsikkotiedostossa `<limits>`):

```
template<class T>
void printMinimumValue()
{
  cout << std::numeric_limits<T>::min();
}
```

Tämä `numeric_limits` -pohjainen työtapa, jolla määritetään tyypeistä riippuvaiset vakiot, voi näyttää kalliilta, mutta se ei ole. Tämä johtuu siitä, että lähdetekstin pitkäveteisyyden epäonnistuu heijastua tuloksena olevan olion koodissa. Itse asiassa `numeric_limits`:n kutsut sisällä oleviin funktioihin eivät luo mitään ohjeita. Nähdäksesi miksi, tutki seuraavaa, joka on ilmeinen tapa toteuttaa `numeric_limits<int>::min`:

```
#include <limits.h>
```

```
namespace std {
  inline int numeric_limits<int>::min() throw ()
  { return INT_MIN; }
}
```

Koska tämä funktio on esitelty avoimena funktiona, siihen tapahtuvat kutsut pitäisi korvata sen rungolla (katso Kohta 33). Tämä koskee vain makroa `INT_MIN`, joka itse on vain yksinkertainen `#define`-direktiivi toteutuksella määritetylle vakiolle. Niinpä vaikka tämän Kohdan alussa oleva `max`-funktio näyttää siltä kuin se suorittaisi funktiokutsun jokaiselle oletuksena olevalle parametrin arvolle, se vain käyttää älykästä tapaa viitata tyypistä riippuvaan vakioon, tässä tapauksessa `INT_MIN`-makron arvoon. Tällaista tehokasta näppäryyttä on runsaasti saatavilla C++-kielen peruskirjastossa. Sinun kannattaa lukea Kohta 49.

Jos palataan `max`-funktioon, kriittinen havainto on, että `max` käyttää samaa (melko tehotonta) algoritmia tuloksen laskemisessa, riippumatta kutsujan välittämien argumenttien määrästä. Tässä funktiossa ei missään kohdassa yritetä selvittää, mitkä parametrit ovat "todellisia" ja mitkä oletusarvoja. Olet sen sijaan valinnut oletusarvon, joka ei mahdollisesti voi vaikuttaa laskemisen kelpoisuuteen sillä algoritmilla, jota käytät. Tämä tekee oletusparametrin arvojen käyttämisestä elinvoimaisen ratkaisun.

Monille funktioille ei ole mitään järkevää oletusarvoa. Oletetaan esimerkiksi, että haluat kirjoittaa funktion, jolla lasketaan enintään viiden `int`-tyyppisen muuttujan keskiarvo. Oletuksena olevia parametriarvoja ei voida käyttää tässä, koska funktion tulos on riippuvainen sille välitettyjen parametrien määrästä: jos kolme arvoa välitettiin, jaat niiden summan kolmella; jos välitettyjä arvoja on viisi, jaat niiden summan viidellä. Lisäksi ei ole olemassa mitään "taikanumeroa", jota voit käyttää oletuksena osoittamaan, että parametri ei itse asiassa ollut asiakkaan välittämä, koska kaikki mahdolliset `int`-tyyppiset muuttujat ovat sallittuja arvoja parametreille. Sinulla ei tässä tapauksessa ole vaihtoehtoja; sinun *täytyy* käyttää kuormitettuja funktioita:

```
double avg(int a);
double avg(int a, int b);
double avg(int a, int b, int c);
double avg(int a, int b, int c, int d);
double avg(int a, int b, int c, int d, int e);
```

Toinen tapaus, jossa sinun pitää käyttää kuormitettuja funktioita, syntyy silloin, kun haluat suorittaa loppuun tietyn tehtävän, mutta käyttämäsi algoritmi riippuu sille annetuista syöttötiedoista. Tämä tapaus tulee yleensä ilmi työskenneltäessä muodostinfunktioiden kanssa: oletuksena oleva muodostinfunktio muodostaa olion alusta alkaen, siinä missä kopiomuodostin muodostaa omansa olemassaolevasta oliosta:

```
// luokka, jolla esitetään luonnollisia lukuja
class Natural {
public:
  Natural(int initValue);
  Natural(const Natural& rhs);

private:
  unsigned int value;

  void init(int initValue);
  void error(const string& msg);
};

inline
void Natural::init(int initValue) { value = initValue; }

Natural::Natural(int initValue)
{
  if (initValue > 0) init(initValue);
  else error("Illegal initial value");
}

inline Natural::Natural(const Natural& x)
{ init(x.value); }
```

Muodostinfunktion, joka vastaanottaa `int`-tyyppisen muuttujan, täytyy suorittaa virheiden tarkistus, mutta kopiomuodostimen ei tarvitse, joten tarvitaan kahta muodostinfunktiota. Tämä tarkoittaa kuormittamista. Huomaa kuitenkin, että molempien funktioiden täytyy sijoittaa alustava arvo uudelle oliolle. Tämä voisi johtaa lähdekoodin kaksinkertaistumiseen kahdessa muodostinfunktiossa, joten taktikoit ongelman kirjoittamalla yksityisen jäsenfunktion `init`, joka sisältää lähdetekstin, joka on yhteistä kahdelle muodostinfunktiolle. Tämä taktiikka - sellaisten kuormitettujen funktioiden käyttäminen, jotka kutsuvat yleistä perustana olevaa funktiota eräisiin tehtäviin - kannattaa muistaa, koska se on säännöllisesti käytännöllinen (katso esimerkki Kohdasta 12).

Kohta 25: Vältä osoittimen ja numeerisen tyypin kuormittamista.

Päivän trivialkysymys: mikä on nolla?

Tarkemmin eriteltynä, mitä tässä tapahtuu?

```
void f(int x);
void f(string *ps);

f(0);                        // kuts. f(int) vai f(string*)?
```

Vastaus on, että 0 on tyypiltään `int` - tarkasti ottaen literaali kokonaislukumuuttuja - joten `f(int)`-funktiota kutsutaan aina. Tässä piilee ongelma, koska tämä ei ole aina sitä mitä halutaan. Tämä tilanne on ainutlaatuinen C++-kielen maailmassa: tässä tilanteessa ajatellaan, että kutsun pitäisi olla monimerkityksinen, mutta kääntäjät eivät ajattele niin.

Olisi mukavaa, jos voisit jotenkin hipsutella tämän ongelman ohi käyttämällä symbolista nimeä, kuten `NULL null`-osoittimille, mutta ongelma näyttää olevan paljon pahempi kuin voisit kuvitella.

Ensimmäinen viehtymyksesi olisi varmaan esitellä vakio nimeltä `NULL`, mutta vakioilla on tyypit, ja mikä olisi `NULL`-vakion tyyppi? Sen täytyy olla yhteensopiva kaikkien osoitintyyppien kanssa, mutta ainoa tyyppi, joka tyydyttää tämän vaatimuksen, on `void*`, etkä voi välittää `void*`-osoittimia tyypitettyihin osoittimiin ilman eksplisiittistä muunnosta. Tämä ei ole pelkästään rumaa, vaan ensi silmäyksellä se ei myöskään ole yhtään parempi kuin alkuperäinen tilanne:

```
void * const NULL = 0;          // mahd. NULL määritys

f(0);                           // kutsuu silti f(int)
f(static_cast<string*>(NULL)); // kutsuu f(string*)
f(static_cast<string*>(0));    // kutsuu f(string*)
```

Kun asiaa ajatellaan toisen kerran, `NULL`-arvon käyttö `void*`-vakiona on hieman parempi kuin se, millä aloitit, koska vältät monimerkityksisyyden, jos käytät vain `NULL`-vakiota null-arvoisten osoittimien osoittamiseen:

```
f(0);                           // kutsuu f(int)
f(NULL);                        // virhe! – väärä tyyppi
f(static_cast<string*>(NULL)); // ok, kutsuu f(string*)
```

Olet nyt ainakin vaihtanut suoritusaikaisen virheen (kutsu "väärään" `f`-funktioon nollalla) käännösajan virheeseen (yritys välittää `void* string*`-tyyppiseen parametriin). Tämä parantaa asioita jonkin verran (katso Kohta 46), mutta muunnos ei silti tyydytä.

Jos ryömit häpeilevästi takaisin esikääntäjäsi luo, huomaat, että se ei todella myöskään tarjoa tietä ulos, koska ilmeiset valinnat tuntuvat olevan

```
#define NULL 0
```

ja

```
#define NULL ((void*) 0)
```

ja ensimmäinen mahdollisuus on vain literaali 0, joka on pohjimmiltaan kokonaislukuvakio (jos muistat alkuperäisen ongelmasi), kun taas toinen mahdollisuus palauttaa sinut takaisin siihen ongelmaan, joka liittyi `void*`-osoittimien välittämiseen tyypitettyihin osoittimiin.

Jos olet päntännyt päähäsi säännöt, jotka koskevat tyyppimuunnosten hallitsemista, tiedät ehkä, että C++-kieli ei pidä `long int` -tyypin muuntamista `int`-tyyppiin yhtään sen huonompana kuin arvosta `long int 0` muuntamista osoittimeen, joka on arvoltaan `null`. Voit hyötyä siitä esittelemällä monimerkityksisyyden `int`/osoitin-kysymykseen, jossa uskoit sen alunperin pitävän ollakin:

```
#define NULL 0L                 // NULL on nyt long int
```

```
void f(int x);
void f(string *p);

f(NULL);                        // virhe! – monimerkityks.
```

Tämä ei kuitenkaan auta sinua yhtään, jos kuormitat `long int` -tyypin ja osoittimen kohdalla:

```
#define NULL 0L

void f(long int x);             // tämä f ottaa nyt long
void f(string *p);

f(NULL);                        // ok, kutsuu f(long int)
```

Tämä on käytännössä ehkä turvallisempaa kuin `NULL`-arvon määrittäminen `int`-tyypillä, mutta kyseessä on pikemminkin tapa siirtää ongelma kuin päästä siitä eroon.

Ongelma voidaan tuhota täysin, mutta se vaatii kieleen tulleen viimeisimmän lisäyksen käyttöä: *jäsenfunktioiden mallien* (niitä kutsutaan usein yksinkertaisemmin *jäsenmalleiksi*). Jäsenfunktiomallit ovat juuri sitä, miltä ne kuulostavat: luokkien sisällä olevia malleja, jotka luovat jäsenfunktioita näille luokille. Haluat `NULL`-tyypin tapauksessa olion, joka toimii kuin ilmaisu `static_cast<T*>(0)` `T`:n jokaiselle tyypille. Tämä viittaa siihen, että `NULL`-tyypin pitäisi olla sen luokan olio, joka sisältää implisiittisen muunnosoperaattorin jokaiselle mahdolliselle osoitintyypille. Muunnosoperaattoreita on paljon, mutta voit pakottaa C++-kielen luomaan ne sinulle jäsenmalleilla:

```
// ensimmäinen kosketus luokkaan saa aikaan
// NULL-arvoisia osoittimia
class NullClass {
public:
  template<class T>                       // luo
    operator T*() const { return 0; }     // operator T* T:n
};                                        // kaikille tyypeil-
                                          // le; funktio
                                          // palauttaa
                                          // null-osoittimen

const NullClass NULL;      // NULL NullClass-tyypin
                           // olio

void f(int x);             // sama kuin alun perin

void f(string *p);         // samat sanat

f(NULL);                   // toimii, muuntaa NULL-tyypp.
                           // string*, kutsuu sitten
                           // f(string*)
```

Tämä on hyvä alkuasetelma, mutta se voidaan jalostaa useilla tavoilla. Ensinnäkään, emme todellakaan tarvitse kuin yhden `NullClass`-olion, joten ei ole mitään syytä antaa luokalle nimeä; voimme vain käyttää nimeämätöntä luokkaa ja tehdä tuosta tyypistä `NULL`-tyyppisen. Toiseksi, niin kauan kuin mahdollistamme `NULL`-arvon muuntamisen minkä tahansa osoittimen tyyppiseksi, meidän pitää myös hoitaa

jäsenten osoittimet. Tämä vaatii toisen jäsenmallin eli sen, jolla `0` konvertoidaan tyypiltään `T C::*` ("osoitin jäseneen, joka on tyyppiä `T` luokassa `C`") kaikille `C`-luokille ja kaikille `T`-tyypeille. (Jos tässä ei mielestäsi ole mitään järkeä, tai jos et ole koskaan kuullut (paljon vähemmän käytetyistä) osoittimista jäseniin, ota rennosti. Jäsenten osoittimet ovat harvinaisia petoja, joita nähdään harvoin luonnossa, ja sinun ei luultavasti koskaan tarvitse olla niiden kanssa tekemisissä. Terminaalisesti utelias voi tutustua Kohtaan 30, jossa jäsenten osoittimet käsitellään yksityiskohtaisemmin.) Lopulta, meidän pitäisi estää asiakkaita ottamasta `NULL`:n osoitetta, koska `NULL`:n ei oleteta käyttäytyvän *osoittimen* lailla, vaan osoittimen *arvon* lailla, ja osoittimen arvoilla (esimerkiksi 0x453AB002) ei ole osoitetta.

Piristetty `NULL`-määritys näyttää tältä:

```
const                           // tämä on const-tyypp. olio
class {
public:
  template<class T>             // muunn. mihin tah. tyyp.
    operator T*() const         // null ei-jäsen
    { return 0; }               // osoittimen...

  template<class C, class T>    // tai minkä tahansa null-
    operator T C::*() const     // arv. jäsenosoittimen...
    { return 0; }

private:
  void operator&() const;       // jonka osoit. ei voida
                                // ottaa (kts. Koh. 27)

} NULL;                         // ja jonka nimi on NULL
```

Tämä on todella katsomisen arvoinen näky, vaikka haluaisit varmaan antaa pikkusormen käytännöllisyydelle antamalla luokalle kuitenkin nimen. Jos et anna, kääntäjän viestit, jotka viittaavat `NULL`:n tyyppiin, ovat mahdottomia ymmärtää.

Tärkeä kohta näissä kaikissa yrityksissä saada aikaan toimiva `NULL` on se, että niistä on apua vain, jos olet *kutsuja*. Jos olet kutsuttujen funktioiden *kirjoittaja*, idioottivarma `NULL` ei auta sinua ollenkaan, koska et voi pakottaa kutsujiasi käyttämään sitä. Vaikka tarjoaisit asiakkaillesi esimerkiksi juuri kehittämämme avaruusajan `NULL`:n, et voi silti estää heitä tekemästä tätä:

```
f(0);                           // kutsuu silti f(int),
                                // koska 0 pn yhä int
```

ja tämä on aivan yhtä ongelmallista, kuin mitä oli tämän Kohdan alussa.

Kuormitettujen funktioiden suunnittelijana on täten parasta välttää numeeristen ja osoitintyyppien kuormittamista, jos vain mahdollisesti voit.

Kohta 26: Suojaa mahdollinen monimerkityksisyys.

Jokaisella on oma filosofiansa. Jotkut ihmiset uskovat *antaa mennä* -oppiin, toiset uskovat jälleensyntymiseen. Jotkut ihmiset jopa uskovat, että COBOL on todellinen ohjelmointikieli. C++-kielellä on myös filosofiansa: sen mielestä mahdollinen monimerkityksisyys ei ole virhe.

Tässä on esimerkki mahdollisesta monimerkityksisyydestä:

```
class B;                          // välitetty esittely
                                  // luokalle B
class A {
public:
  A(const B&);                    // A voidaan
                                  // muodostaa B:stä
};

class B {
public:
  operator A() const;             // B voidaan
                                  // muuntaa A:ksi
};
```

Näiden luokkien esittelyssä ei ole mitään väärin - ne voivat olla olemassa samassa ohjelmassa ilman pienintäkään ongelmaa. Katso kuitenkin mitä tapahtuu, kun yhdistät nämä luokat funktiolla, joka ottaa A-olion, mutta sille välitetään itse asiassa B-olio:

```
void f(const A&);

B b;

f(b);                             // virhe! – monimerkityks.
```

Kun kääntäjät näkevät f-funktioon tapahtuvan kutsun, ne tietävät, että niiden pitää jotenkin saada aikaan olio, joka on tyyppiä A, vaikka niillä on hallussaan B-tyyppinen olio. On olemassa kaksi täysin yhtä hyvää tapaa tehdä tämä. Ensinnäkin luokan A muodostinfunktiota voitaisiin kutsua; tämä muodostaisi uuden A-olion käyttäen b:tä argumenttina. b voitaisiin toisaalta muuntaa A-olioksi kutsumalla luokassa B asiakkaan määrittämää muunnosoperaattoria. Koska näitä kahta työtapaa pidetään yhtä hyvinä, kääntäjät kieltäytyvät valitsemasta niiden väliltä.

Voisit tietenkin käyttää tätä ohjelmaa jonkin aikaa ilman, että koskaan törmäisit monimerkityksisyyteen. Tämä on mahdollisen monimerkityksisyyden salakavala vaara. Se voi olla ohjelmassa piilevänä pitkiä aikoja, huomaamattomana ja passiivisena, kunnes koittaa päivä, jolloin pahaa aavistamaton ohjelmoija tekee jotain, joka todella *on* monimerkityksistä, ja silloin käynnistyy varsinainen sirkus. Tämä aiheuttaa hämmentävän mahdollisuuden, että voit esittää kirjaston, jota voit kutsua monimerkityksisesti ilman, että olet tietoinen siitä, että teet niin.

Samanlainen monimerkityksisyyden muoto voi ilmetä kielen perusmuunnoksissa - et tarvitse edes luokkia:

```
void f(int);
void f(char);

double d = 6.02;

f(d);                           // virhe! – monimerkityks.
```

Pitäisikö `d` muuntaa tyyppiin `int` vai `char`? Muunnokset ovat yhtä hyviä, joten kääntäjät eivät toimi tuomareina. Tämä ongelma voidaan onneksi ohittaa käyttämällä eksplisiittistä muunnosta:

```
f(static_cast<int>(d));         // toimii, kutsuu f(int)
f(static_cast<char>(d));        // toimii, kutsuu f(char)
```

Moniperintä (katso Kohta 43) on täynnä mahdollisuuksia monimerkityksisyyteen. Suoraviivaisin esimerkki on, että periytetty luokka perii saman jäsennimen useammasta kuin yhdestä kantaluokasta:

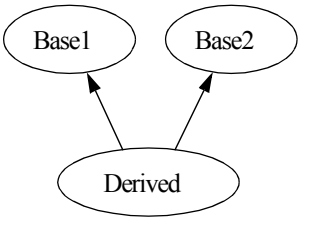
```
class Base1 {
public:
  int doIt();
};

class Base2 {
public:
  void doIt();
};

class Derived: public Base1,     // periytetty ei esittele
               public Base2 {    // funktiota nimeltä doIt
  ...
};

Derived d;

d.doIt();                       // virhe! – monimerkityks.
```



Kun `Derived`-luokka perii kaksi funktiota, joilla on sama nimi, C++-kieli ei inahdakaan; tällä hetkellä monimerkityksisyys on vain potentiaalista. `doIt`-funktion kutsu pakottaa kääntäjän kuitenkin kohtaamaan asian, ja sitä paitsi, jos et eksplisiittisesti tee monimerkityksettömäksi kutsua määrittelemällä, minkä kantaluokan funktion haluat, kutsu johtaa virheeseen:

```
d.Base1::doIt();                // ok, kutsuu Base1::doIt
d.Base2::doIt();                // ok, kutsuu Base2::doIt
```

Tämä tuskin järkyttää kovin monia ihmisiä, mutta tosiasia, että saannin rajoitukset eivät tule esille, on saanut useamman kuin yhden muuten niin pasifistisen sielun mietiskelemään selvästi militäärisiä toimia:

```
class Base1 { ... };            // sama kuin edellä

class Base2 {
private:
  void doIt();                  // tämä funktio on nyt
};                              // yksityinen

class Derived: public Base1, public Base2
{ ... };                        // sama kuin edellä

Derived d;

int i = d.doIt();               // virhe! – edell. monim.!
```

Kutsu `doIt`-funktioon on edelleen monimerkityksinen, vaikka vain luokan `Base1` funktio on saatavissa! Tosiasia, että vain `Base1::doIt` palauttaa arvon, jota voidaan käyttää `int`-tyyppisen arvon alustamiseen, on myös asiaankuulumaton - kutsu on edelleen monimerkityksinen. Jos haluat tehdä tämän kutsun, sinun *täytyy* yksinkertaisesti määrittää, minkä luokan `doIt`-funktio on se, jonka haluat.

Kuten on myös asia niiden kaikkien C++-kielen sääntöjen kanssa, jotka eivät alkuaan ole välittömästi tajuttavissa, on olemassa hyvä syy, miksi saatavuuden rajoituksia ei oteta lukuun silloin, kun monimerkityksellisiä viittauksia poistetaan moniperiytyneistä jäsenistä. Kaikki johtaa tähän: luokan jäsenen saatavuuden muuttamisen ei pitäisi koskaan muuttaa ohjelman tarkoitusta.

Oletetaan esimerkiksi, että saatavuuden rajoitukset otettiin huomioon edellisessä esimerkissä. Silloin `d.doIt()`-ilmaisu johtaisi kutsuun `Base1::doIt`, koska `Base2`-luokan versio ei ollut saavutettavissa. Oletetaan nyt, että `Base1` muutettiin niin, että sen versio `doIt`-funktiosta on suojattu sen sijasta, että se olisi julkinen, ja `Base2` muutettiin niin, että sen versio on julkinen sen sijasta, että se olisi yksityinen.

Yhtäkkiä saman `d.doIt()`-ilmaisun tuloksena olisi *täysin erilainen funktiokutsu*, vaikka ei kutsuvaa koodia eikä funktioita ollut muutettu! *Tätä* voisi kutsua epäintuitiiviseksi, ja kääntäjillä ei olisi mitään keinoa päästää edes varoitusta. Kun tutkitaan vaihtoehtoja, voi olla, että päätät, että moniperiytyneisiin jäseniin kohdistuneiden viittausten monimerkityksisyyden poistaminen ei ole niin järjenvastaista kuin alunperin ajattelit.

Kun otetaan huomioon, että kaikki nämä eri tavat ovat olemassa ohjelmien ja kirjastojen kirjoittamiseen, idätelleen mahdollista monimerkityksisyyttä, mitä pitää hyvän ohjelmistojen kehittäjän tehdä? Sinun pitää ensi sijassa pitää sitä silmällä. On melkein mahdotonta penkoa esille kaikkia mahdollisen monimerkityksisyyden lähteitä, varsinkin, kun ohjelmoijat yhdistävät kirjastoja, jotka on kehitelty itsenäisesti (katso myös Kohta 28). Ymmärtämällä tilanteet, jotka usein johtavat mahdolliseen monimerkityksisyyteen, olet kuitenkin paremmassa asemassa vähentääksesi sen olemassaoloa ohjelmissa, joita suunnittelet ja kehität.

Kohta 27: Estä eksplisiittisesti niiden implisiittisesti luotujen jäsenfunktioiden käyttö, joita et halua.

Oletetaan, että haluat kirjoittaa luokkamallin `Array`, jonka luodut luokat käyttäytyvät täysin samalla tavalla kuin sisäänrakennetut C++-taulukot, paitsi että ne suorittavat raja-arvojensa tarkistuksen. Yksi työtavan ongelmista, jonka tulet kohtaamaan, on, kuinka sijoitukset kielletään `Array`-olioiden välillä, koska sijoitus ei ole sallittua C++-kielen taulukoille:

```
double values1[10];
double values2[10];

values1 = values2;              // virhe!
```

Tämä ei useimmille funktioille olisi ongelma. Jos et haluaisi sallia funktiota, et yksinkertaisesti sijoittaisi sitä luokkaan ollenkaan. Sijoitusoperaattori on kuitenkin yksi noista arvokkaista jäsenfunktioista, joita C++-kieli, tuo alati avulias palvelija, kirjoittaa sinulle, jos laiminlyöt niiden kirjoittamisen itse (katso Kohta 45). Mitä tehdään silloin?

Ratkaisu on esitellä funktio, tässä tapauksessa `operator=`, *yksityisenä*. Kun esittelet jäsenfunktion eksplisiittisesti, estät kääntäjiä luomasta omia versioita, ja kun teet funktiosta yksityisen, estät ihmisiä kutsumasta sitä.

Skeema ei kuitenkaan ole idioottivarma; jäsen- ja ystäväfunktiot voivat silti kutsua yksityistä funktiotasi. *Paitsi,* jos olet tarpeeksi älykäs, etkä *määritä* funktiota. Jos tällöin huomaamatta kutsut funktiota, saat linkityksen aikana virheilmoituksen (katso Kohta 46).

`Array`-taulukon mallin määritys voisi alkaa tähän tyyliin:

```
template<class T>
class Array {
private:
  // Älä määritä tätä funktiota
  Array& operator=(const Array& rhs);

  ...
};
```

Jos asiakas yrittää nyt suorittaa sijoituksia `Array`-olioihin, kääntäjät tekevät yritykset tyhjiksi, ja jos yrittäisit huomaamatta suorittaa sijoituksen jäsen- tai ystäväfunktiossa, linkkerisi älähtäisi.

Tästä esimerkistä ei kannata olettaa, että tämä Kohta pätee vain sijoitusoperaattoreihin. Se ei päde. Se pätee kaikkiin kääntäjän luomiin funktioihin, jotka kuvattiin Kohdassa 45. Huomaat käytännössä, että sijoituksen ja kopiomuodostimen käytöksen samankaltaisuudet tarkoittavat melkein aina sitä, että kun haluat estää yhden käytön, haluat estää myös toisen käytön.

Kohta 28: Osioi globaali nimiavaruus.

Globaalin näkyvyysalueen suurin ongelma on siinä, että niitä on vain yksi. Suuressa ohjelmistoprojektissa on yleensä suuri joukko ihmisiä, jotka sijoittavat nimensä tähän yksittäiseen näkyvyysalueeseen, ja tämä johtaa poikkeuksetta nimiristiriitoihin. Otsikkotiedosto `library1.h` voisi esimerkiksi määrittää joukon vakioita, mukaanlukien seuraavan:

```
const double LIB_VERSION = 1.204;
```

Sama koskee `library2.h`-otsikkotiedostoa:

```
const int LIB_VERSION = 3;
```

Ei tarvita paljon oivalluskykyä huomaamaan, että odotettavissa on ongelma, kun ohjelma yrittää sisällyttää sekä `library1.h`-tiedoston että `library2.h`-tiedoston. Valitettavasti, poislukien mielessä kiroamisen, vihamielisen postin lähettämisen kirjaston kirjoittajille ja otsikkotiedostojen muokkaamisen, kunnes nimien ristiriitaisuudet on eliminoitu, on vähän, mitä voit tehdä tämänkaltaiselle ongelmalle.

Voit kuitenkin armahtaa ne köyhät sielut, jotka on huijattu ottamaan *sinun* kirjastosi käyttöön. Olet jo luultavasti suunnitellut jonkinlaisen toivottavasti yksilöllisen etuliitteen jokaisen globaalin symbolisi eteen, mutta toki sinun täytyy myöntää, että tuloksena olevat tunnisteet ovat vähemmän miellyttävää katseltavaa.

Parempi ratkaisu on käyttää C++-kielen nimiavaruutta (`namespace`). Tiivistelmänä perusolemuksesta `namespace` on vain hieno tapa sallia sinun käyttää tuntemiasi etuliitteitä, joita rakastat, ilman, että ihmisten tarvitsee katsella niitä koko ajan. Joten tämän sijasta,

```
const double sdmBOOK_VERSION = 2.0;   // tässä kirjastossa,
                                      // jokainen symboli al-
class sdmHandle { ... };              // kaa "sdm"

sdmHandle& sdmGetHandle();         // katso Kohdasta 47
                                   // miksi haluat määr.
                                   // funktion tähän tyyliin
```

kirjoitat tämän:

```
namespace sdm {
  const double BOOK_VERSION = 2.0;
  class Handle { ... };
  Handle& getHandle();
}
```

Asiakkaat pääsevät näin nimiavaruudessasi oleviin symboleihin millä tahansa kolmesta tavallisesta tavasta: tuomalla kaikki nimiavaruuden symbolit näkyvyysalueeseen, tuomalla yksittäiset symbolit näkyvyysalueeseen tai kelpuuttamalla symbolin eksplisiittisesti kertakäyttöä varten. Tässä on muutamia esimerkkejä:

```
void f1()
{
  using namespace sdm;          // tee kaikki sdm:n symbol.
                                // saataville ilman määritystä
                                // tässä näkyvyysalueessa
  cout << BOOK_VERSION;         // ok, selviää arv.
                                // sdm::BOOK_VERSION
  ...
  Handle h = getHandle();       // ok, Handle selviää
                                // sdm::Handle, getHandle
  ...                           // selviää sdm::getHandle
}
void f2()
{
  using sdm::BOOK_VERSION;      // tee vain BOOK_VERSION
                                // saat. ilman kelpuutusta
                                // tässä näkyvyysalueessa
  cout << BOOK_VERSION;         // ok, selviää arv.
                                // sdm::BOOK_VERSION
  ...
  Handle h = getHandle();       // virhe! kumpikaan Handle
                                // tai getHandle ei
  ...                           // tuotu tähän näkyvyysal.
}
void f3()
{
  cout << sdm::BOOK_VERSION;    // ok, tekee BOOK_VERSION
                                // saataville vain
  ...                           // käyttöä varten
  double d = BOOK_VERSION;      // virhe! BOOK_VERSION ei
                                // ole näkyvyysalueessa
  Handle h = getHandle();       // virhe! kumpikaan Handle
                                // tai getHandle ei
  ...                           // tuotu tähän näkyvyysal.
}
```

Yksi nimiavaruuksien mukavimmista piirteistä on se, että mahdollinen monimerkityksisyys ei ole virhe. Tästä on tuloksena, että voit tuoda saman symbolin useammasta kuin yhdestä nimiavaruudesta, ja voit silti elää huoletonta elämää (olettaen tietysti, että et koskaan käytä tuota symbolia). Jos sinulla on esimerkiksi tarve käyttää `sdm`-nimiavaruuden lisäksi tätä nimiavaruutta,

```
namespace AcmeWindowSystem {
  ...
```

```
  typedef int Handle;
  ...
}
```

voisit käyttää ilman ristiriitaa sekä `sdm`- että `AcmeWindowSystem`-nimiavaruutta, olettaen, että et koskaan viittaisi `Handle`-symboliin. Jos viittaisit siihen, sinun täytyisi eksplisiittisesti kertoa, minkä nimiavaruuden `Handle`-tyypin olisit halunnut.

```
void f()
{
  using namespace sdm;                // tuo sdm-symbolit
  using namespace AcmeWindowSystem;   // tuo Acme-symbolit

  ...                                 // viittaa vap. sdm
                                      // ja Acme-symboleihin
                                      // muihin kuin Handle

  Handle h;                           // virhe! mikä Handle?

  sdm::Handle h1;                     // ok, ei monimerkit.

  AcmeWindowSystem::Handle h2;        // samoin ei monimerk.

  ...
}
```

Aseta tämä vastakkain perinteisen otsikkotiedostoon perustuvan ajattelun kanssa, jossa pelkkä otsikkotiedostojen `sdm.h` ja `acme.h` sisällyttäminen saisi kääntäjät valittamaan `Handle`-symbolin monista määrityksistä.

Nimiavaruudet lisättiin C++-kieleen suhteellisen myöhään standardointileikissä, joten ne eivät ehkä sinusta ole kovin tärkeitä ja voit elää ilman niitä. Et voi. Et voi, koska melkein kaikki peruskirjastossa oleva (katso Kohta 49) on `std`-nimiavaruuden sisällä. Tämä voi tuntua sinusta pieneltä yksityiskohdalta, mutta se vaikuttaa sinuun erittäin suoralla tavalla: C++-kieli tukee nyt nimiavaruuksien avulla miellyttävän näköisiä tarkennittomia otsikkonimiä, kuten `<iostream>`, `<string>` ja niin edelleen. Jos haluat lukea yksityiskohtia, katso Kohta 49.

Koska nimiavaruudet esiteltiin suhteellisen äskettäin, kääntäjäsi eivät ehkä vielä tue niitä. Jos näin on, ei ole silti syytä saastuttaa globaalia nimiavaruutta, koska voit approksimoida nimiavaruudet struktuureilla (`struct`). Teet sen luomalla `struct`-struktuurin, joka tallentaa globaalit nimet sijoittaen ne sitten tämän struktuurin sisälle staattisina jäseninä:

```
// nimiavaruutta emuloivan struktuurin määritys
struct sdm {
  static const double BOOK_VERSION;
```

```
  class Handle { ... };
  static Handle& getHandle();
};
const double sdm::BOOK_VERSION = 2.0;   // staattisen
                                        // tietojäsenen
                                        // pakollinen
                                        // määritys
```

Nyt kun ihmiset haluavat pääsyn globaaleihin nimiisi, he yksinkertaisesti varustavat ne struktuurin nimen etuliitteellä:

```
void f()
{
  cout << sdm::BOOK_VERSION;

  ...

  sdm::Handle h = sdm::getHandle();

  ...
}
```

Jos globaalilla tasolla ei ole nimiristiriitoja, kirjastosi asiakkaista voi ehkä olla raskasliikkeistä käyttää täysin päteviä nimiä. Onneksi on olemassa tapa, jolla voit antaa heille oman näkyvyysalueen ja samalla myös hylätä ne.

Varusta tyyppiesi nimille `typedef`-komennot, jotka poistavat tarpeen ekplisiittiselle näkyvyysalueen luomiselle. Varusta siis (globaali) `typedef`-komento tyyppinimelle `T`, joka on nimiavaruuden tyyppisessä rakenteessa `S` niin, että `T` on synonyymi `S::T`-funktiolle:

```
typedef sdm::Handle Handle;
```

Sisällytä rakenteesi jokaiselle (staattiselle) `X`-oliolle (globaali) viittaus `X`, joka alustetaan `S::X:`n arvolla:

```
const double& BOOK_VERSION = sdm::BOOK_VERSION;
```

Rehellisesti sanoen, kun olet lukenut Kohdan 47, ajatus siitä, että määrität epäpaikallisen staattisen olion, kuten `BOOK_VERSION`, saa sinut luultavasti pahoinvoivaksi. (Haluat varmaan korvata tällaiset oliot Kohdassa 47 kuvatuilla funktioilla.)

Funktioita kohdellaan melko samalla tavalla kuin olioita, mutta vaikka viittausten määrittäminen funktioihin on sallittua, lähdetekstisi tulevat ylläpitäjät vihaavat sinua paljon vähemmän, jos otat funktioiden sijasta käyttöön osoittimet:

```
sdm::Handle& (* const getHandle)() =    // getHandle on
  sdm::getHandle;                       // const-tyyppinen
                                        // osoitin (kts.
                                        // Kappale 21)
                                        // sdm::getHandle
```

Huomaa, että `getHandle` on *const*-tyyppinen osoitin. Et varmastikaan halua antaa asiakkaiden määrittää sitä osoittamaan johonkin muuhun kuin `sdm::getHandle`:en, ethän?

(Jos olet innokas tietämään, kuinka määritetään viittaus funktioon, tämän pitäisi virkis-tää muistiasi:

```
sdm::Handle& (&getHandle)() =  // getHandle on viittaus
  sdm::getHandle;              // sdm::getHandle-funktioon
```

Ajattelen henkilökohtaisesti, että tämä on aika hienoa, mutta on olemassa syy, jonka takia et ehkä koskaan ole nähnyt tätä aikaisemmin. (Lukuunottamatta tapaa, jolla ne on alustettu, viittaukset funktioihin ja `const`-tyyppiset osoittimet funktioihin käyttäytyvät identtisesti, ja funktioiden osoittimet ovat helpommin luettavissa ja ymmärrettävissä.)

Näillä `typedef`-komennoilla ja viittauksilla asiakkaat, jotka eivät kärsi globaalisten nimien ristiriidoista, voivat käyttää pätemätöntä tyyppiä ja olio-nimiä, kun taas asiakkaat, joilla on nimiristiriitoja, voivat jättää ottamatta huomioon `typedef`-komennot ja viittausmääritykset sekä käyttää täysin päteviä nimiä. On epätodennäköistä, että kaikki asiakkaasi haluavat käyttää pikakirjoitusnimiä, joten sinun kannattaa varmistaa, että sijoitat `typedef`-komennot ja viittaukset eri otsikkotiedostoon, kuin siihen, joka sisältää nimiavaruutta (`namespace`) emuloivan `struct`-struktuurisi.

`struct`-struktuurit ovat mukava approksimaatio `namespace`-nimiavaruuksille, mutta todelliseen juttuun on vielä pitkä matka. Ne ovat riittämättömiä monella eri tavalla, joista yksi ilmeisin on niiden operaattorikäsittely. Operaattoreita, jotka on yksinkertaisesti sanoen määritetty `struct`-struktuurien `static`-tyyppisinä jäsenfunktioina, voidaan pyytää avuksi vain funktiokutsun avulla, ei koskaan sen luonnollisen sisäänrakennetun kieliopin avulla, jota operaattoreiden on suunniteltu tukevan:

```
// määritä nimiavaruutta emuloiva strukt., joka sisältää
// Widget-luokkien tyypit ja funktiot. Widget-oliot
// tukevat lisäämistä operator+-funktion kautta
struct widgets {
  class Widget { ... };

  // katso Kohdasta 21 miksi paluuarvo on tyyp. const
  static const Widget operator+(const Widget& lhs,
                                const Widget& rhs);

  ...
};

// yritys asettaa globaaleja (pätemättömiä) nimiä Widget-
// oliolle ja operator+-funktiolle kuten kuvattu edellä

typedef widgets::Widget Widget;
```

```
const Widget (* const operator+)(const Widget&,   // virhe!
                                 const Widget&);  // operator+
                                                  // ei voi olla
                                                  // osoitinnimi

Widget w1, w2, sum;

sum = w1 + w2;                      // virhe! tässä
                                    // näkyvyysalueessa
                                    // ei ole määritetty
                                    // mitään operator+-
                                    // funktiota

sum = widgets::operator+(w1, w2);   // sallittu, mutta
                                    // tuskin "luonnolli-
                                    // nen" kielioppi
```

Tällaisten rajoitusten pitäisi kannustaa sinua omaksumaan todellisten nimiavaruuksien käyttö niin pian, kuin kääntäjäsi vain tekevät asiasta käytännöllisen.