Sekalaista

Eräät tehokkaan C++-kielen ohjelmoinnin ohjeistot uhmaavat sopivaa luokittelua. Tässä kirjan osassa tämänkaltaiset ohjeistot menevät yöpuulle. Ei silti, että se vähentäisi niiden tärkeyttä. Jos aiot kirjoittaa tehokkaita ohjelmia, sinun täytyy ymmärtää se, mitä kääntäjät tekevät sinulle (sinulle?) selän takana, kuinka varmistat sen, että epäpaikalliset staattiset oliot alustetaan ennen kuin niitä käytetään, mitä voit odottaa normaalilta kirjastolta, ja mistä voit löytää oivalluksia kielen perustana olevaan työtapafilosofiaan. Laajennan näitä kysymyksiä, ja enemmänkin, tämän kirjan viimeisessä osassa.

Kohta 45: Tunnista, mitä funktioita C++-kieli kirjoittaa ja kutsuu hiljaisesti.

Milloin tyhjä luokka ei ole tyhjä luokka? Silloin kun C++-kieli läpäisee sen. Jos niitä ei esitellä itse, ajattelevaiset kääntäjäsi esittelevät omat versionsa kopiomuodostimesta, sijoitusoperaattorista, tuhoajafunktiosta ja parista osoite-operaattorista. Jos et lisäksi esittele yhtään muodostinfunktiota, ne esittävät myös oletusmuodostinfunktion. Kaikki nämä funktiot tulevat olemaan julkisia. Toisin sanoen, jos kirjoitat tämän:

```
class Empty{};
```

kyseessä on sama, kuin jos olisit kirjoittanut sen näin:

```
class Empty {
public:
  Empty();                          // oletusmuodostin
  Empty(const Empty& rhs);          // kopiomuodostin

  ~Empty();                         // tuhoajafunkt. – katso
                                    // alapuolelta, onko
                                    // se virtuaalinen
  Empty&
  operator=(const Empty& rhs);      // sijoitusoperaattori

  Empty* operator&();               // address-of-operaatt.
  const Empty* operator&() const;
};
```

Nämä funktiot generoidaan nyt vain tarvittaessa, mutta ne eivät ole kovin tarpeellisia. Seuraava koodi aiheuttaa jokaisen funktion luonnin:

```
const Empty e1;                 // oletusmuodostinfunktio;
                                // tuhoajafunktio
Empty e2(e1);                   // kopiomuodostin
e2 = e1;                        // sijoitusoperaattori
Empty *pe2 = &e2;               // address-of-
                                // operaattori (ei-const)
const Empty *pe1 = &e1;         // address-of-
                                // operaattori (const)
```

Silloin kun kääntäjät kirjoittavat funktiot sinulle, mitä funktiot sitten tekevät? No, oletusmuodostinfunktio ja tuhoajafunktio eivät itse asiassa tee mitään. Ne vain sallivat sinun luoda ja tuhota luokan olioita. (Ne tarjoavat myös toteuttajille mukavan paikan sijoittaa koodia, jonka suoritus huolehtii "kulissien takana tapahtuvasta" käyttäytymisestä - katso Kohta 33.) Huomaa, että luotu tuhoajafunktio on epävirtuaalinen (katso Kohta 14), paitsi jos se on luokka, joka periytyy kantaluokasta, joka itse esittelee virtuaalituhoajafunktion. Oletuksena olevat osoiteoperaattorit pelkästään palauttavat olion osoitteen. Nämä funktiot määritetään tehokkaasti tähän tyyliin:

```
inline Empty::Empty() {}
inline Empty::~Empty() {}
inline Empty * Empty::operator&() { return this; }
inline const Empty * Empty::operator&() const
{ return this; }
```

Mitä tulee kopiomuodostimeen ja sijoitusoperaattoriin, virallinen sääntö on tämä: oletuksena oleva kopiomuodostin (sijoitusoperaattori) suorittaa luokan epästaattisten tietojäsenten jäsenkohtaisen kopiomuodostuksen (sijoituksen). Tämä tarkoittaa sitä, että jos `m` on `C`-luokan epästaattinen tietojäsen ja `C` ei esittele kopiomuodostinta (sijoitusoperaattoria), `m` kopiomuodostetaan (sijoittamaan) käyttämällä kopiomuodostinta (sijoitusoperaattoria), joka on määritetty `T`-luokalle, jos sellainen on. Jos ei ole, tätä sääntöä sovelletaan rekursiivisesti `m`:n tietojäsenille, kunnes kopiomuodostin (sijoitusoperaattori) tai sisäänrakennettu tyyppi (eli `int`, `double`, osoitin jne.) on löydetty. Oletuksena on, että sisäänrakennettujen tyyppien oliot periytyvät muista luokista. Tämä sääntö pätee periytyvyyshierarkian jokaiselle tasolle, joten käyttäjän määrittelemiä kopiomuodostimia ja sijoitusoperaattoreita kutsutaan kaikilla niillä tasoilla, joilla ne on esitelty.

Toivottavasti tämä on vesiselvää sinulle.

Jos ei ole selvää, tässä on esimerkki. Tutki `NamedObject`-mallin määritystä, jonka ilmentymät ovat luokkia, jotka sallivat sinun yhdistää nimiä olioihin:

```
template<class T>
class NamedObject {
public:
  NamedObject(const char *name, const T& value);
  NamedObject(const string& name, const T& value);

  ...

private:
  string nameValue;
  T objectValue;
};
```

Koska `NamedObject`-oliot esittelevät ainakin yhden muodostinfunktion, kääntäjät eivät käännä oletusmuodostinfunktioita, mutta koska luokat jättävät esittelemättä kopiomuodostimet tai sijoitusoperaattorit, kääntäjät luovat nämä funktiot (jos niitä tarvitaan).

Tutki seuraavaa kutsua kopiomuodostimeen:

```
NamedObject<int> no1("Smallest Prime Number", 2);

NamedObject<int> no2(no1);     // kutsuu kopiomuodostinta
```

Kääntäjiesi luoman kopiomuodostimen täytyy alustaa `no2.nameValue` ja `no2.objectValue` käyttämällä vastaavasti `no1.nameValue` ja `no1.objectValue` arvoja. `nameValue`:n tyyppi on merkkijono, ja merkkijonolla on kopiomuodostin (voit todentaa sen tutkimalla peruskirjastossa olevaa merkkijonoa - katso Kohta 49), joten `no2.nameValue` alustetaan kutsumalla merkkijonon kopiomuodostinta argumenttinaan `no1.nameValue`. Toisaalta, `NamedObject<int>::objectValue`:n tyyppi on `int` (koska `T` on tyypiltään `int` mallin tälle instantioinnille), ja `int`-tyypeille ei ole määritetty kopiomuodostinta, joten `no2.objectValue` tullaan alustamaan kopioimalla ne `no1.objectValue`-muuttujasta.

Kääntäjän luoma sijoitusoperaattori `NamedObject<int>`:lle käyttäytyisi samalla tavalla, mutta kääntäjän luomat sijoitusoperaattorit käyttäytyvät yleisesti ottaen kuvaamallani tavalla vain silloin, kun tuloksena oleva koodi sekä on sallittua että sillä on järkeenkäypä mahdollisuus olla järkevää. Jos jompikumpi näistä testeistä epäonnistuu, kääntäjät kieltäytyvät luomasta luokallesi `operator=`-funktiota, ja tulet vastaanottamaan kääntämisen aikana ihanaa diagnostiikkaa.

Oletetaan esimerkiksi, että `NamedObject` olisi määritetty tällä tavalla, jossa `nameValue` on *viittaus* merkkijonoon ja `objectValue` on *`const`*-tyyppinen `T`:

```
template<class T>
class NamedObject {
public:
  // tämä kopiomuodostin ei enää ota vastaan const-nimeä,
  // koska name-Value on nyt viittaus ei-const-
  // tyypp. merkkijonoon. Kopiomuodostin on poissa, koska
  // meidän pitää viitata merkkijonoon.
  NamedObject(string& name, const T& value);

  ...                           // sama kuin ed., olettaen
                                // operator= on esitelty
private:
  string& nameValue;            // tämä on nyt viittaus
  const T objectValue;          // tämä on nyt const
};
```

Tutki seuraavaksi mitä tässä tapahtuisi:

```
string newDog("Persephone");
string oldDog("Satch");

NamedObject<int> p(newDog, 2);    // kun kirjoitan tätä,
                                  // Persephone-koirallamme
                                  // on kohta hänen toinen
                                  // syntymäpäivänsä

NamedObject<int> s(oldDog, 29);   // perhekoira Satch
                                  // (lapsuudestani)
                                  // olisi 29, jos se olisi
                                  // yhä hengissä

p = s;                            // mitä tapahtuisi p:
                                  // tietojäseniin?
```

`p.nameValue` viittaa ennen sijoitusta johonkin `string`-olioon ja `s.nameValue` myös `string`-olioon, vaikkakaan ei samaan. Kuinka sijoituksen pitäisi vaikuttaa `p.nameValue`-muuttujaan? Pitäisikö `p.nameValue`:n viitata sijoituksen jälkeen `string`-olioon, johon viitataan `s.nameValue`-muuttujalla, eli pitäisikö itse viittausta muuttaa? Jos näin on, luodaan uutta, koska C++-kieli ei tarjoa tapaa antaa viittauksen viitata eri olioon. Pitäisikö `string`-olion, johon `p.nameValue` viittaa, tulla vaihtoehtoisesti muutetuksi, vaikuttaen täten muihin olioihin, jotka säilyttävät osoittimet tai viittaukset tuohon merkkijonoon, eli olioihin, jotka eivät suoraan ole mukana sijoituksessa? Pitäisikö kääntäjän luoman sijoitusoperaattorin tehdä tämä?

Kun C++-kieli joutuu kasvotusten tällaisten sanaleikkien kanssa, se kieltäytyy kääntämästä koodia. Jos haluat tukea sijoitusta luokassa, joka sisältää viittausjäsenen, sinun täytyy määrittää sijoitusoperaattori itse. Kääntäjät käyttäytyvät samalla tavoin luokille, jotka sisältävät `const`-tyyppisiä jäseniä (kuten `objectValue` edellä mainitussa muutetussa luokassa); `const`-tyyppisiä jäseniä ei ole sallittua muuttaa, joten kääntäjät ovat epävarmoja siitä, kuinka kohdella niitä implisiittisesti luodun sijoitusfunktion aikana. Kääntäjät kieltäytyvät lopulta luomasta sijoitusoperaattoreita periytetyille luokille, jotka periytyvät kantaluokista, jotka esittelevät standardin sijoi-

tusoperaattorin `private`-tyyppisenä. Kääntäjien luomien sijoitusoperaattoreiden oletetaan käsittelevän myös kantaluokan osat periytetyille luokille (katso Kohta 16), mutta kun ne tekevät niin, niiden ei tietenkään pitäisi pyytää avuksi niitä jäsenfunktioita, joita periytetyllä luokalla ei ole oikeutta kutsua.

Kaikki tämä puhe kääntäjän luomista funktioista johtaa kysymykseen siitä, mitä tehdä, jos haluaa estää näiden funktioiden käytön? Tämä tarkoittaa sitä, että mitä jos et esimerkiksi tarkoituksella esittele `operator=`-funktiota, koska et koskaan halua sallia sijoittamista luokassasi oleviin olioihin? Ratkaisu tähän pieneen pulmalliseen kysymykseen on Kohdan 27 aihe. Jos haluat tutustua keskusteluun usein sivuutetusta vuorovaikutuksesta osoitinjäsenten ja kääntäjän luomien kopiomuodostimien ja sijoitusoperaattoreiden välillä, tarkista Kohta 11.

Kohta 46: Suosi kääntämisenaikaisia ja linkittämisenaikaisia virheitä verrattuna suoritusaikaisiin virheisiin.

Paitsi muutamissa tilanteissa, jotka aiheuttavat C++-kielen muodostuvan poikkeuksen (eli muisti loppuu - katso Kohta 7), suoritusaikaisen virheen käsite on vieras C++-kielelle samoin kuin C-kielellekin. Kielessä ei tapahdu mitään alivuodon, ylivuodon, nollalla jakamisen, ei raja-arvojen virheellisyyden tarkistuksen tunnistusta. Kun ohjelmasi pääsee kääntäjän ja linkkerisi ohi, vastuu siirtyy sinulle - seuraamuksille ei ole mitään turvaverkkoa. Kuten laskuvarjolla hyppääminen, jotkut ihmiset ovat innoissaan tämänkaltaisista asioista, toiset taas jähmettyvät kauhusta. Filosofian takana oleva motivaatio on tietenkin tehokkuus: ohjelmat ovat pienempiä ja nopeampia ilman suoritusajan tarkistusta.

Smalltalkin ja LISPin kaltaiset kielet tutkivat yleisesti ottaen vähemmän virheitä kääntämisen ja linkittämisen aikana, mutta ne sisältävät voimakkaan suoritusajan järjestelmän, jotka saavat virheet kiinni suorituksen aikana. Nämä kielet ovat C++-kielestä poiketen melkein aina tulkattuja, ja maksat suorituksesta koituvan rangaistuksen niiden tarjoamasta ylimääräisestä joustavuudesta.

Älä koskaan unohda, että olet ohjelmoimassa C++-kielellä. Vaikka Smalltalk/LISP-filosofia tuntuu sinusta vetoavalta, unohda se. Paljon puhutaan siitä, että pidetään kiinni puolueen linjasta, ja tässä tapauksessa se tarkoittaa pakenemista suoritusajan virheistä. Aina kun voit, siirrä virheidentunnistus suoritusajasta linkittämishetkeen, tai ideaalisesti ajatellen kääntämisen hetkeen.

Tällainen menetelmäoppi maksaa osinkonsa, ei pelkästään ohjelman koossa ja nopeudessa, vaan myös luotettavuudessa. Jos ohjelmasi pääsee kääntäjän ja linkkerisi läpi ilman virheviestien ilmaantumista, voit olla varma, että ohjelmassasi ei ole kääntäjän tai linkkerin tunnistettavissa olevia virheitä, piste. (Toinen mahdollisuus on tietenkin, että kääntäjissäsi tai linkkerissäsi on ohjelmointivirheitä, mutta älkäämme masentuko myöntämällä tällaisia mahdollisuuksia.)

Suoritusaikaisten virheiden kanssa tilanne on erilainen. Kun ohjelmasi ei luo mitään suoritusaikaisia virheitä tietyn suorituksen aikana, kuinka voit olla varma, että se ei luo virheitä eri suorituksen aikana, silloin kun teet asiat eri järjestyksessä, käytät eri tietoa, tai suoritat pidemmän tai lyhyemmän aikavälin? Voit testata ohjelmaasi, kunnes olet naamaltasi sininen, mutta et silti koskaan käy läpi kaikkia mahdollisuuksia. Tästä on tuloksena, että virheiden tunnistaminen suorituksen aikana on yksinkertaisesti vähemmän turvallista kuin niiden kiinnisaaminen kääntämisen tai linkittämisen aikana.

Usein, kun teet suhteellisen pieniä muutoksia työssäsi, voit saada kääntämisen aikana kiinni sen, mikä voisi muuten olla suoritusajan virhe. Tämä koskee säännöllisesti uusien tyyppien lisäämistä ohjelmaasi. Oletetaan esimerkiksi, että olet kirjoittamassa luokkaa, jolla esitetään päivämääriä. Ensimmäinen leikkaus voisi näyttää tältä:

```
class Date {
public:
  Date(int day, int month, int year);

  ...

};
```

Jos aikoisit toteuttaa tämän muodostinfunktion, yksi ongelmista, jonka tulisit kohtaamaan, olisi päivän ja kuukauden arvojen järkevyyden testaaminen. Katsokaamme, kuinka voimme eliminoida tarpeen kelpuuttaa arvo, joka on välitetty kuukautena.

Yksi ilmeinen työtapa on työllistää lueteltu tyyppi kokonaisluvun sijasta:

```
enum Month { Jan = 1, Feb = 2, ... , Nov = 11, Dec = 12 };

class Date {
public:
  Date(int day, Month month, int year);

  ...

};
```

Valitettavasti tämä ei vie kovin pitkälle, koska luettelotyyppejä ei tarvitse alustaa:

```
Month m;
Date d(22, m, 1857);          // m on määrittelemätön
```

Tästä on tuloksena, että `Date`-muodostinfunktion täytyisi silti kelpuuttaa kuukausiparametrin arvo.

Kun haluat saavuttaa tarpeeksi turvallisuutta, jotta olet vapautettu suoritusajan tarkistuksista, sinun täytyy käyttää luokkaa, joka esittää kuukausia, ja sinun täytyy varmistaa, että vain oikeita kuukausia luodaan:

```
class Month {
public:
  static const Month Jan() { return 1; }
  static const Month Feb() { return 2; }
  ...
  static const Month Dec() { return 12; }

  int asInt() const             // mukavuuden takia tee
  { return monthNumber; }       // mahdolliseksi Month-luo-
                                // kan muuttam. int-tyyp.

private:
  Month(int number): monthNumber(number) {}

  const int monthNumber;
};

class Date {
public:
  Date(int day, const Month& month, int year);
  ...
};
```

Tämän työtavan useat näkökulmat yhdistyvät saaden sen toimimaan omalla tavallaan. Ensinnäkin, `Month`-muodostinfunktio on tyypiltään `private`. Tämä estää asiakkaita luomasta uusia kuukausia. Ainoat käytettävissä olevat kuukaudet ovat niitä, jotka ovat `Month`-olion staattisten jäsenfunktioiden palauttamia, sekä siitä johtuvat kopiot. Toiseksi, jokainen `Month`-olio on `const`-tyyppinen, joten sitä ei voida muuttaa. (Muuten houkutus muuntaa tammikuu heinäkuuksi voisi osoittautua musertavaksi, ainakin pohjoisilla leveysasteilla.) Ainoa tapa saada `Month`-olio lopulta on kutsua funktiota tai kopioida olemassa oleva Month-olio (implisiittisen `Month`-kopiomuodostimen kautta - katso Kohta 45). Tämä mahdollistaa `Month`-olion käytön missä tahansa ja mihin aikaan tahansa; ei tarvitse murehtia sitä, että käyttää sitä vahingossa ennen kuin se on alustettu. (Kohdassa 47 selvitetään, miksi tämä voisi muuten olla ongelma.)

Asiakkaan on näiden luokkien avulla kaikkea muuta kuin mahdollista määrittää pätemätön kuukausi. Se olisi täysin mahdotonta, ilman seuraavaa inhottavaa asiaa:

```
Month *pm;                    // määritä alustamaton ost
Date d(1, *pm, 1997);         // arghhh! käytä sitä!
```

Tämä koskee kuitenkin viittaamista pois alustamattomasta osoittimesta. jonka tulokset ovat määrittelemättömiä. (Katso Kohta 3, jos haluat lukea mielipiteeni määrittelemättömästä käyttäytymisestä.) En valitettavasti tiedä mitään tapaa estää tai tunnistaa tämänkaltaista harhaoppisuutta. Jos kuitenkin oletamme, että tätä ei koskaan tapahdu, tai jos emme välitä siitä, kuinka ohjelmamme käyttäytyy, jos se tekee niin, `Date`-muodostinfunktio voi hallita selväjärkisyyden tarkistuksen sen `Month`-parametrin kohdalla. Muodostinfunktion täytyy silti toisaalta tarkistaa `day`-parametrin laillisuus - kuinka monta päivää on syyskuussa, huhtikuussa, heinäkuussa ja marraskuussa?

Tämä `Date`-esimerkki korvaa suoritusajan tarkistukset kääntämisen ajan tarkistuksilla. Ihmettelet varmaan, milloin on mahdollista käyttää linkittämisen ajan tarkistuksia. Tosiasia on, että ei kovin usein. C++-kieli käyttää linkkeriä varmistaakseen, että tarvittavat funktiot määritetään tarkasti kerran (katso Kohdan 45 kuvaus siitä mitä "tarvitaan" funktioon). Se myös käyttää linkkeriä varmistaakseen, että staattiset oliot (katso Kohta 47) on määritetty tarkalleen kaksi kertaa. Sinulla on taipumus käyttää linkkeriä samalla tavalla. Esimerkiksi Kohdassa 27 kuvataan, kuinka linkkerin tarkistukset voivat tehdä käytännölliseksi välttää tahallaan eksplisiittisesti esittelemäsi funktion määrittämistä.

Älä kuitenkaan innostu liikaa. On epäkäytännöllistä eliminoida tarve *kaikelle* suoritusajan virhetarkastukselle. Esimerkiksi minkä tahansa ohjelman, joka vastaanottaa interaktiivista syöttöä, tulee suorittaa syöttötiedon tarkistus. Vastaavasti luokan, joka toteuttaa taulukoita, jotka suorittavat raja-arvojen tarkistuksen (katso Kohta 18), täytyy kelpuuttaa taulukon indeksi raja-arvoon joka kerta, kun taulukkoon on pääsy. Tarkistusten siirtäminen suoritusajasta kääntämisen tai linkittämisen aikaiseksi on joka tapauksessa aina kannattava tavoite, ja sinun kannattaa jahdata tuota tavoitetta aina, kun se on käytännöllistä. Tästä on palkintona, että ohjelmasi ovat pienempiä, nopeampia ja luotettavampia.

Kohta 47: Varmista, että epäpaikalliset staattiset oliot alustetaan ennen kuin niitä käytetään.

Olet jo aikuinen, joten minun ei tarvitse kertoa sinulle, että on tyhmänrohkeaa käyttää alustamatonta oliota. Koko käsite voi itse asiassa tuntua absurdilta; muodostinfunktiot varmistavat, että oliot alustetaan ennen kuin ne luodaan, *eikö niin?*

Vastaus on kyllä ja ei. Kaikki toimii hienosti erityisessä käännösyksikössä (eli lähdetiedostossa), mutta asiat mutkistuvat, kun yhdessä käännösyksikössä olevan olion alustus riippuu eri käännösyksikössä olevan toisen olion arvosta *ja* tuo toinen olio itsekin vaatii alustuksen.

Oletetaan esimerkiksi, että olet kirjoittanut kirjaston, joka tarjoaa tiedostojärjestelmän abstraktion, ja sisältää mahdollisesti sellaisen ominaisuuden, kuten Internetin tiedostojen näyttäminen niin kuin ne olisivat paikallisia tiedostoja. Koska kirjastosi saa maailman näyttämään yhdeltä tiedostojärjestelmältä, voisit luoda kirjastosi nimiavaruuteen erikoisolion, `theFileSystem` (katso Kohta 28), jota asiakkaasi voisivat käyttää aina, kun heidän täytyy olla vuorovaikutuksessa kirjastosi tarjoaman tiedostojärjestelmän abstraktion kanssa:

```
class FileSystem { ... };       // tämä luokka on nyt
                                // kirjastossasi

FileSystem theFileSystem;       // tämä on olio,
                                // jonka kanssa kirjaston
                                // asiakkaat keskustelevat
```

Koska `theFileSystem` edustaa jotain monimutkaista, ei ole yllätys, että sen muodostus on sekä epätriviaalia että elintärkeää; `theFileSystem`-olion käyttö ennen muodostamista olisi saanut aikaan erittäin määrittelemätöntä käyttäytymistä.

Oletetaan seuraavaksi, että joku kirjastosi asiakas luo luokan tiedostojärjestelmäsi hakemistoille. Heidän luokkansa käyttää luonnollisesti `theFileSystem`-oliota:

```
class Directory {               // kirjaston asiakk. luoma
public:
  Directory();
  ...
};

Directory::Directory()
{
  luo Directory-olio pyytämällä avuksi
  theFileSystemin funktioita;
}
```

Oletetaan lisäksi, että tämä asiakas päättää luoda erillisen globaalin `Directory`-olion tilapäisille tiedostoille:

```
Directory tempDir;              // hakemisto tilapäisille
                                // tiedostoille
```

Alustusjärjestyksen ongelma tulee nyt ilmiselväksi: paitsi jos `theFileSystem` on alustettu ennen `tempDir`-oliota, `tempDir`-olion muodostinfunktio yrittää käyttää `theFileSystem`-oliota ennen kuin se on alustettu. Mutta oliot `theFileSystem` ja `tempDir` olivat eri ihmisten eri aikoina eri tiedostoissa luomia. Kuinka voit olla varma, että `theFileSystem` tullaan luomaan ennen `tempDir`-oliota?

Tämänkaltainen kysymys saa alkunsa aina, kun sinulla on *epäpaikallisia staattisia olioita*, jotka on määritetty eri käännösyksiköissä, ja joiden oikea käyttäytyminen on riippuvainen siitä, että ne on alustettu erityisessä järjestyksessä. Epäpaikalliset oliot ovat olioita, jotka on

- määritetty globaalissa tai nimiavaruuden näkyvyysalueessa (toisin sanoen `theFileSystem` ja `tempDir`),
- esitelty staattisina luokassa tai
- määritetty tiedoston näkyvyysalueessa staattisina.

"Ei-paikallisille staattisille olioille" ei ikävä kyllä ole olemassa pikakirjoitustermiä, joten sinun kannattaa tottua tähän hankalanpuoleiseen fraasiin.

Et halua, että ohjelmasi käyttäytyminen on riippuvainen eri käännösyksiköissä olevien ei-paikallisten staattisten olioiden alustusjärjestyksestä, koska et voi hallita tuota järjestystä. Toistan tämän. *Et ehdottomasti pysty hallitsemaan eri käännösyksiköissä olevien ei-paikallisten staattisten olioiden alustusjärjestystä*.

On järkevää ihmetellä, miksi asia on näin.

Asia on näin, koska "sopivan" järjestyksen, jossa epäpaikalliset staattiset oliot alustetaan, määrittäminen on työlästä. Erittäin työlästä. Sairaan työlästä. Kaikkein yleisimmässä muodossaan - jossa ovat useat käännösyksiköt ja epäpaikalliset staattiset oliot, jotka on luotu implisiittisten malli-instantiointien avulla (jotka itsekin voivat saada alkunsa implisiittisten malli-instantiointien avulla) - ei ole pelkästään mahdotonta määrittää oikeaa alustusjärjestystä, ei edes kannata etsiä erikoistapauksia, joissa *on* mahdollista määrittää oikea järjestys.

Kaaos-teoriassa on periaate, joka tunnetaan nimellä "Perhosefekti". Tämä periaate vakuuttelee, että pienikin perhosten siipien havisemisen aikaansaama ilmakehän häiriö toisessa maanosassa voi johtaa syvällisiin muutoksiin kaukana olevien paikkojen sääkaavoissa. Se vakuuttelee vielä tiukkapipoisemmin, että eräissä järjestelmätyypeissä minuuttien levottomuus syötössä voi johtaa radikaaleihin muutoksiin tulosteissa.

Ohjelmistojärjestelmien kehitys voi esittää oman perhosefektinsä. Eräät järjestelmät ovat ylen haavoittuvaisia vaatimustensa yksityiskohdista, ja pienet muutokset vaatimuksissa voivat merkitsevästi vaikuttaa siihen helppouteen, jossa järjestelmä voidaan toteuttaa. Kohdassa 29 kuvataan esimerkiksi, kuinka implisiittisen konversion määrityksen muuttaminen `String`-tyyppisestä `char*`-tyyppiseksi muotoon

`String`-tyyppisestä `const char*` -tyyppiseksi tekee mahdolliseksi korvata hidas tai virhealtis funktio nopealla ja turvallisella funktiolla.

Ongelma varmistettaessa, että epäpaikalliset staattiset oliot alustetaan ennen käyttöä, on vastaavasti herkkä niille yksityiskohdille, jotka haluat saavuttaa. Jos sen sijasta, että vaadit pääsyä epäpaikallisiin staattisiin olioihin, olet halukas tyytymään pääsyyn olioihin, jotka *toimivat* epäpaikallisten staattisten olioiden tapaisesti (paitsi alustuksen päänsäryissä), vaikea ongelma poistuu. Jäljellä on sen sijaan ongelma, joka on niin helppo ratkaista, että sitä ei edes kannata kutsua ongelmaksi.

Tekniikka - joka joskus tunnetaan nimellä *Singleton-kaava* - on itse kansantajuisuus. Siirrät ensinnäkin jokaisen epäpaikallisen staattisen olion omaan funktioonsa, jossa esittelet sen `static`-tyyppisenä. Seuraavaksi funktio palauttaa viittauksen sisältämäänsä olioon. Asiakkaat kutsuvat funktiota sen sijasta, että viittaisivat olioon. Korvaat toisin sanoen epäpaikalliset, staattiset oliot olioilla, jotka ovat `static`-tyyppisiä funktioiden sisällä.

Tämän työtavan lähtökohta on huomio, että vaikka C++-kieli ei sano paljon mitään silloin, kun epäpaikallinen, staattinen olio alustetaan, se yksilöityy aika tarkasti silloin, kun funktion sisällä oleva staattinen olio (eli paikallinen staattinen olio) alustetaan. Joten, jos korvaat suorat pääsyt epäpaikallisiin staattisiin olioihin kutsuilla funktioihin, jotka palauttavat viittaukset niiden sisällä oleviin paikallisiin staattisiin olioihin, voit olla varma, että viittaukset, jotka saat takaisin funktioista, viittaavat alustettuihin olioihin. Bonuksena, jos kutsut koskaan funktiota, joka emuloi epäpaikallista staattista oliota, sinulle ei koskaan muodostu kustannuksia olion muodostamisesta ja tuhoamisesta. Samaa ei voida sanoa todellisista ei-paikallisista staattisista olioista.

Tässä on tekniikka, joka pätee sekä `theFileSystem`- ja `tempDir`-olioihin:

```
class FileSystem { ... };       // sama kuin ennen

FileSystem& theFileSystem()     // tämä funktio korvaa
{                               // theFileSystem-olion

  static FileSystem tfs;        // määritä ja alusta pai-
                                // kallinen staatt. olio
                                // (tfs = "the file system")

  return tfs;                   // palauttaa viitt. siihen
}

class Directory { ... };        // sama kuin ennen

Directory::Directory()
{
  sama kuin ennen, mutta viittaukset theFileSystem-olioon
on korvattu viittauksilla theFileSystem()-funktioon;
}

Directory& tempDir()            // tämä funktio korvaa
{                               // tempDir-olion
```

```
    static Directory td;          // määritä/alusta paikall.
                                  // staattinen olio
    return td;                    // palauta viittaus siihen
  }
```

Tämän muutetun järjestelmän asiakkaat ohjelmoivat täsmälleen samalla tavalla, kuin mihin ovat tottuneet, paitsi että nyt ne viittaavat `theFileSystem()`- ja `tempDir()`-funktioihin. Tämä tarkoittaa sitä, että ne viittaavat vain funktioihin, jotka palauttavat viittaukset näihin olioihin, ne eivät koskaan viittaa itse olioihin.

Tämän skeeman sanelemat, viittauksen palauttavat funktiot ovat aina yksinkertaisia; määritä ja alusta paikallinen staattinen olio rivillä 1, palauta se rivillä 2. Siinä kaikki. Koska ne ovat niin yksinkertaisia, sinua voi houkuttaa esitellä ne avoimina funktioina (`inline`). Kohdassa 33 selvitetään, että C++-kielen määritykseen tehdyt viimeisimmät tarkistukset tekevät tästä täysin kelvon toteutusstrategian, mutta siinä selvitetään myös, miksi haluat varmasti tarkastaa kääntäjäsi yhteensopivuuden standardin tämän näkökulman kanssa, ennen kuin otat sen käyttöön. Jos kokeilet sitä kääntäjällä, joka ei vielä ole yhtäpitävä standardin asiaankuuluvien osien kanssa, otat riskin saada useita kopioita sekä saantifunktiosta että sen sisällä määritetystä staattisesta oliosta. Tämä voi saada aikuisen ohjelmoijan kyyneliin.

Tässä ei ole mitään taikuutta. Jotta tämä tekniikka olisi tehokas, järkevän alustusjärjestyksen määritys pitää olla mahdollinen olioillesi. Jos määrität niin, että olion A pitää olla alustettu ennen oliota B, ja teet myös A:n alustuksesta riippuvaisen siitä, että B on jo alustettu, joudut vaikeuksiin, ja rehellisesti sanoen ansaitset sen. Jos kuitenkin vetäydyt ujona pois tällaisista patologisista tilanteista, tässä Kohdassa kuvattu skeema palvelisi sinua aika mukavasti.

Kohta 48: Ota huomioon kääntäjän varoitukset.

Monet ohjelmoijat jättävät rutiininomaisesti kääntäjän varoitukset huomioonottamatta. Jos ongelma olisi vakava, sehän olisi virhe, eikö niin? Tämän tapainen ajattelu voi olla suhteellisen harmitonta muissa kielissä, mutta C++-kielessä voin melkein lyödä vetoa, että kääntäjän kirjoittajilla on parempi ote siihen mitä tapahtuu kuin sinulla. Tässä on esimerkiksi virhe, jonka jokainen tekee ennemmin tai myöhemmin:

```
class B {
public:
  virtual void f() const;
};

class D: public B {
public:
  virtual void f();
};
```

`D::fD::`-funktion tarkoituksena on määrittää virtuaalifunktio `B::f` uudelleen, mutta kyseessä on virhe: `f` on `const`-tyyppinen jäsenfunktio, joka on `B`-luokassa, mutta sitä ei ole esitelty `const`-tyyppisenä `D`-luokassa. Yksi tietämäni kääntäjä antaa tällaisen virheilmoituksen:

```
warning: D::f() hides virtual B::f()
```

Liian monet kokemattomat ohjelmoijat vastaavat tähän viestiin sanomalla itselleen, "Tottakai `D::f`-funktio piilottaa `B::f`-funktion - senhän *oletetaankin* tekevän niin!" Väärin. Se, mitä tämä kääntäjä yrittää kertoa sinulle, on, että `B`-luokassa esiteltyä `f`-funktiota ei ole esitelty uudelleen `D`-luokassa, se on piilotettu kokonaan (katso Kohta 50, jos haluat kuvauksen miksi). Jos jätät huomioimatta tämän kääntäjän antaman varoituksen, se tulee melkein varmasti johtamaan ohjelman virheelliseen käyttäytymiseen, josta on seurauksena paljon virheenjäljitystä sen syyn löytämiseksi, jonka tämä kääntäjä tunnisti heti.

Kun saat kokemusta oman kääntäjäsi virheilmoituksista, opit tietenkin ymmärtämään, mitä eri ilmoitukset tarkoittavat (mikä valitettavasti usein on eri asia, kuin mitä ne *näyttävät* tarkoittavan). Kun olet saanut kokemusta, voi olla, että tulee olemaan koko joukko varoituksia, jotka jätät huomioonottamatta. Tämä käy, mutta on tärkeää varmistaa, että ennen kuin hylkäät varoituksen, ymmärrät tarkasti, mitä se yrittää kertoa sinulle.

Kun käsittelemme varoitusaihetta, palauta vielä mieleen, että varoitukset ovat luonnostaan riippuvaisia toteutuksesta, joten ei ole hyvä ajatus heittäytyä hutiloivaksi ohjelmoinnissasi luottaen kääntäjiin, että ne huomaisivat virheet puolestasi. Esimerkiksi edellä mainittu funktion piilottava koodi pääsee toisen (mutta paljon käytetyn) kääntäjän läpi ilman, että kukaan protestoi. Kääntäjien on tarkoitus kääntää C++-kielinen koodi suoritettavaan muotoon, mutta niiden ei ole tarkoitus toimia turvaverkkona. Haluatko tämänkaltaista turvallisuutta? Ohjelmoi sitten Ada-kielellä.

Kohta 49: Tutustu peruskirjastoon.

C++-kielen peruskirjasto on iso. Erittäin iso. Äärettömän iso. Kuinka iso? Esitän asian tällä tavalla: erittely vie tilaa yli 300 tiukasti pakattua sivua C++-standardissa, mutta se ei toki sisällä C-kielen peruskirjastoa, joka sisältyy C++-kirjastoon - "viitteenä". (Tämä on todella termi, jota he käyttävät.)

Isompi ei tietenkään aina tarkoita samaa kuin parempi, mutta tässä tapauksessa isompi *on* parempi, koska iso kirjasto sisältää paljon toiminnallisuutta. Mitä enemmän peruskirjastossa on toiminnallisuutta, sitä enemmän voit tukeutua toiminnallisuuteen, kun kehität sovelluksiasi. C++-kirjasto ei sisällä *kaikkea* (tuki moniajolle ja graafisiin käyttöliittymiin puuttuu huomioitavasti), mutta se sisältää paljon. Voit nojautua siihen lähes kokonaan.

Ennen kuin teen yhteenvedon siitä, mitä kirjastossa on, minun pitää kertoa sinulle, kuinka kirjasto on järjestetty. Koska kirjastossa on niin paljon tavaraa, on olemassa järkeenkäypä mahdollisuus, että sinä (tai joku kaltaisesi) voit valita luokan tai funktion nimen, joka on sama kuin mikä löytyy peruskirjastosta. Suojellakseen sinua tuloksena olevilta nimien ristiriidoilta, lähes kaikki peruskirjastossa oleva on pesiytyneenä `std`-nimiavaruuteen (katso Kohta 28). Mutta tämä johtaa uuteen ongelmaan. On olemassa triljoonia ja ziljoonia rivejä C++-koodia, joka luottaa sen pseudostandardin kirjaston toiminnallisuuteen, joka on ollut käytössä vuosia, eli toiminnallisuuteen, joka on esitelty otsikkotiedostoissa kuten `<iostream.h>`, `<complex.h>`, `<limits.h>` ja niin edelleen. Tätä olemassaolevaa ohjelmistoa ei ole suunniteltu käyttämään nimiavaruutta, ja olisi häpeä, jos peruskirjaston kietominen `std`-nimiavaruudella aiheuttaisi olemassaolevan koodin toimivuuden katkeamisen. (Rikkimenneen koodin kirjoittajat käyttäisivät melko varmasti koko lailla karkeampaa kieltä kuin "häpeä" kuvaamaan tunteitaan siitä, että kirjastollinen matto on vedetty heidän jalkojensa alta.)

Kiinnittäen huomion suitsuttavista ohjelmoijista koostuvan mellakoivan ryhmän tuhoavaan voimaan, standardointikomitea päätti luoda uudet otsikkonimet `std`-nimiavaruuteen kiedotuille komponenteille. Algoritmi, jonka he valitsivat luomaan uudet otsikkonimet, on aivan yhtä triviaali kuin tulokset, jotka se tuottaa epävakaana: `.h` olemassaolevista C++-otsikoista yksinkertaisesti tiputettiin. Joten `<iostream.h>` -tiedostosta tuli `<iostream>`, `<complex.h>`-tiedostosta tuli `<complex>` ja niin edelleen. C-kielen otsikoille käytettiin samaa algoritmia, mutta c-kirjain on lisätty jokaisen nimen eteen. Täten C-kielen `<string.h>`-tiedostosta tuli `<cstring>`, `<stdio.h>` -tiedostosta tuli `<cstdio>` ja niin edelleen. C++-kielen vanhoja otsikkoja kehotettiin viimeisenä temppuna *välttämään* (eli ne luetellaan niin, että niitä ei tueta enää), mutta vanhoja C-kielen otsikkoja ei kehotettu välttämään (jotta säilytettäisiin yhteensopivuus). Käytännössä kääntäjien kauppiailla ei ole kiihotinta hylätä heidän asiakkaittensa laillista ohjelmistoa, joten voit odottaa, että vanhoja C++-otsikoita tuetaan vielä monta vuotta.

Tämä on sitten käytännöllisesti katsoen C++-kielen otsikon tilanne:

- Vanhoja C++-otsikkonimiä kuten `<iostream.h>` tullaan melko varmasti tukemaan, vaikka ne eivät olekaan virallisessa standardissa. Tällaisten otsikkojen sisältö *ei* ole `std`-nimiavaruudessa.

- Uudet C++-otsikkonimet kuten `<iostream>` sisältävät saman perustoiminnallisuuden kuin niiden vastaavat vanhat otsikkotiedostot, mutta otsikkojen sisältö on `std`-nimiavaruudessa. (Eräiden kirjastokomponenttien yksityiskohdat muutettiin standardoinnin aikana, joten tarkkaa vastiketta vanhassa C++-otsikossa oleville ja niille, jotka ovat uudessa otsikossa, ei välttämättä ole olemassa.)
- Normaaleja C-otsikoita, kuten `<stdio.h>`, tuetaan edelleen. Tällaisten otsikkojen sisältö ei ole `std`-nimiavaruudessa.
- Uusilla C++-otsikoilla, jotka vastaavat C-kirjastojen toiminnallisuutta, on sen tapaisia nimiä kuten `<cstdio>`. Ne sisältävät saman sisällön, kuin niiden vastaavat vanhat C-otsikot, mutta sisältö on `std`-nimiavaruudessa.

Kaikki tämä näyttää aluksi hieman oudolta, mutta tottuminen ei ole ollenkaan niin vaikeaa. Suurin haaste on siinä, että kaikki merkkijono-otsikot pysyvät selkeinä: `<string.h>` on vanha C-otsikko `char*`-pohjaisille merkkijonojen manipulointifunktioille, `<string>` on `std`-nimiavaruuteen kiedottu C++-otsikko uusille merkkijonoluokille (katso alla), ja `<cstring>` on `std`-nimiavaruuteen kiedottu versio vanhasta C-kielen otsikosta. Jos voit hallita tämän (ja tiedän että voit), loppuosa kirjastosta on helppo.

Seuraava asia, joka sinun pitää tietää peruskirjastosta on, että melkein kaikki siinä oleva on malli (template). Tutki vanhaa ystävääsi iostream-kirjastoa. (Jos sinä ja iostream-kirjastot ette vielä ole ystäviä, siirry Kohtaan 2 nähdäksesi, miksi sinun pitäisi viljellä ystävyyttä.) Iostream-kirjastot auttavat sinua käsittelemään merkkijonovirtoja, mutta mikä on merkki? Onko se `char`? Vai `wchar_t`? Unicode-merkki? Jokin muu monitavuinen merkki? Ei ole ilmiselvää oikeaa vastausta, joten kirjasto antaa sinulle mahdollisuuden valita. Kaikki virtausluokat ovat itse asiassa luokkamalleja, ja määrität merkin tyypin silloin kun instantioit virtausluokan. Peruskirjasto määrittää esimerkiksi, että `coutin` tyyppi on `ostream`, mutta `ostream` on todella typedef-komento `basic_ostream<char>`-mallille.

Vastaavat pohdinnat pätevät peruskirjaston useimpiin muihin luokkiin. `string` ei ole luokka, se on luokkamalli: type-parametri määrittelee jokaisen merkkijonoluokan merkkien tyypin. `complex` ei ole luokka, se on luokkamalli: tyyppi-parametri määrittää jokaisessa `complex`-luokassa olevien todellisten ja kuvitteellisten komponenttien tyypin. `vector` ei ole luokka, se on luokkamalli. Ja niin edelleen.

Peruskirjaston malleja ei voi paeta, mutta jos olet tottunut työskentelemään vain virtausten ja `char`-tyyppisten merkkijonojen kanssa, voit useimmiten jättää ne huomioimatta. Tämä johtuu siitä, että kirjasto määrittää kirjaston näiden komponenttien `typedef`-tyypit `char`-tyyppien instantioinneille, sallien täten sinun jatkaa ohjelmointia olioiden kuten `cin`, `cout`, `cerr` ehdoilla, ja tyyppien `istream`, `ostream`, `string` kanssa ilman, että sinun tarvitsee huolehtia siitä, että `cin`-tyypin oikea tyyppi on `basic_istream<char>` ja `string`-tyypin `basic_string<char>`.

Monet peruskirjaston komponentit sijaitsevat malleissa enemmän, kuin mitä tässä esitetään. Tutki jälleen näennäisen suoraviivaista käsitettä kuten merkkijono. Se voidaan tottakai parametroida perustuen siihen merkkijonotyyppiin, jonka se tallentaa, mutta erilaiset merkkijonojoukot eroavat yksityiskohdiltaan, esimerkiksi erityiset loppumismerkit, tehokkain tapa kopioida niistä koostuvia taulukoita ja niin edelleen. Tällaiset erityistuntomerkit tunnetaan standardissa *erikoisuuksina*, ja ne määritetään `string`-instantioinneissa lisänä olevalla mallin parametrillä. `string`-tyyppiset oliot tulevat lisäksi melko varmasti suorittamaan muistin dynaamista varaamista ja vapauttamista, mutta on olemassa paljon erilaisia tapoja suorittaa tuo tehtävä (katso Kohta 10). Mikä on paras? Saat itse valita: `string`-tyyppinen malli vastaanottaa `Allocator`-parametrin, ja `Allocator`-tyyppisiä olioita käytetään `string`-olioiden käyttämän muistin varaamiseen ja vapauttamiseen.

Tässä on täysimittainen esittely `basic_string`-mallille ja `string`-tyyppiselle typedef-komennolle, joka rakentuu siitä; voit löytää tämän (tai jotain sitä vastaavaa) `<string>`-otsikosta:

```
namespace std {

  template<class charT,
           class traits = char_traits<charT>,
           class Allocator = allocator<charT> >
    class basic_string;

  typedef basic_string<char> string;

}
```

Huomaa, kuinka `basic_string`-tyypillä on oletusarvot `traits`- ja `Allocator`-parametreille. Tämä on tyypillinen peruskirjasto. Se tarjoaa joustavuutta niille, jotka tarvitsevat sitä, mutta "tyypilliset" asiakkaat, jotka haluavat vain tehdä "normaalin" asian, voivat jättää huomioonottamatta monimuotoisuuden, joka mahdollistaa joustavuuden. Toisin sanoen, jos haluat `string`-olioita, jotka toimivat enemmän tai vähemmän kuin C-kielen merkkijonot, voit käyttää `string`-olioita ja olla edelleen iloisesti tietämätön siitä, että käytät itse asiassa olioita, joiden tyyppi on `basic_string<char, char_traits<char>, allocator<char> >`.

Yleensä näin on. Joskus sinun täytyy hieman kurkistaa konepellin alle. Kohdassa 34 keskustellaan esimerkiksi luokan esittelemisen eduista silloin, kun sen määritystä ei anneta, ja huomautetaan, että seuraava on väärä tapa esitellä `string`-tyyppi:

```
class string;                  // tämä kääntyy, mutta
                               // et halua tehdä niin
```

Laitetaan nimiavaruuden pohdinnat syrjään hetkeksi, sillä todellinen ongelma tässä on se, että `string` ei ole luokka, se on `typedef`. Olisi mukavaa, jos pystyisit ratkaisemaan ongelman tällä tavalla:

```
typedef basic_string<char> string;
```

mutta se ei käänny. Kääntäjäsi ihmettelevät, "Mikä on tämä `basic_string`, josta puhut?" vaikka ne ehkä ääntäisivät kysymyksen aika lailla eri lailla. Kun haluat esitellä merkkijonon, esittelisit ensin kaikki mallit, joista se on riippuvainen. Jos pystyisit tekemään sen, se näyttäisi tältä

```
template<class charT> struct char_traits;

template<class T> class allocator;

template<class charT,
         class traits = char_traits<charT>,
         class Allocator = allocator<charT> >
  class basic_string;

typedef basic_string<char> string;
```

Et voi kuitenkaan esitellä merkkijonoa. Sinun ei ainakaan pitäisi pystyä tekemään niin. Tämä johtuu siitä, että kirjastojen toteuttajien sallitaan esitellä `string` (tai jotain muuta `std`-nimiavaruudessa olevaa) eri tavalla, kuin mitä on määritetty standardissa, niin kauan, kun tulos tarjoaa standardia mukailevaa käyttäytymistä. `basic_string`-toteutus voisi esimerkiksi lisätä neljännen malliparametrin, mutta tuon parametrin oletusarvon täytyisi pyytää avuksi koodia, joka toimii standardin määräämällä koristelemattomalla tavalla niin kuin `basic_stringin`.

Lopputulos? Älä yritä esitellä `string`-tyyppiä manuaalisesti (tai mitään muuta peruskirjaston osaa). Lisäät sen sijaan vain vastaavan otsikkotiedoston, esimerkiksi `<string>`.

Kun sinulla on hallussasi nämä taustatiedot otsikoista ja malleista, olemme siinä tilanteessa, että voimme katsastaa C++-kielen peruskirjaston pääkomponentit:

- **Perus-C-kirjasto**. Se on yhä tallella, ja voit yhä käyttää sitä. Pieniä juttuja on säädetty siellä sun täällä, mutta se on aikeineen ja tarkoituksineen sama C-kirjasto, joka on ollut olemassa jo monia vuosia.
- **Iostream-kirjastot**. Verrattuna "perinteisiin" iostream-toteutuksiin, ne ovat nyt malleissa, ja sen periytyvyyshierarkiaa on muutettu, siihen on lisätty kyky muodostaa poikkeus, ja se on päivitetty tukemaan `string`-tyyppisiä muuttujia (`stringstream`-luokkien avulla) ja kansainvälistymistä (locale-muuttujien avulla - katso alla). Silti lähes kaikki, mitä olet tottunut odottamaan iostream-kirjastolta, on edelleen olemassa. Se tukee varsinkin virtauspuskureita, muotoilijoita, käsittelijöitä ja tiedostoja, sekä olioita kuten `cin`, `cout`, `cerr` ja `clog`. Tämä tarkoittaa sitä, että voit kohdella merkkijonoja ja tiedostoja virtoina, ja sinulla on laaja mahdollisuus hallita virtauksen käyttäytymistä, mukaan lukien puskurointi ja muotoilu.

- **Merkkijonot**. `string`-oliot suunniteltiin eliminoimaan tarve käyttää `char*`-pohjaisia osoittimia useimmissa ohjelmissa. Ne tukevat odotettuja toimintoja (eli merkkijonojen yhdistäminen, muuttuja-aikainen pääsy yksittäisiin merkkeihin `operator[]`-funktion avulla ja niin edelleen), ne ovat muutettavissa `char*`-pohjaisiksi muuttujiksi niin, että yhteensopivuus perintökoodin kanssa säilyy, ja ne hoitavat muistinhallinnan automaattisesti. Eräät merkkijonototeutukset työllistävät viittausten laskennan, mikä voi johtaa parempaan suoritukseen (sekä ajassa että tilassa) kuin `char*`-pohjaiset merkkijonot.

- **Säiliöt**. Lopeta omien perussäiliöluokkien kirjoittaminen! Kirjasto tarjoaa tehokkaat toteutukset vektori-malleista (ne toimivat kuten dynaamisesti laajennettavissa olevat taulukot), listat (kaksoislinkitys), jono, pinot, deque-muuttujat, kartat, sarjat ja bittisarjat. Valitettavasti kirjastossa ei ole hash-tyyppisiä tauluja (monet jälleenmyyjät kauppaavat niitä laajennuksina), mutta tätä kompensoi se, että `string`-muuttujat ovat säiliöitä. Tämä on tärkeää, sillä se tarkoittaa sitä, että kaiken, mitä voit tehdä säiliöllä (katso alla), voit tehdä myös `string`-tyypeillä.

  Mikä se on? Haluat tietää, kuinka minä *tiedän*, että kirjastototeutukset ovat tehokkaita? Helppoa: kirjasto määrittää jokaisen luokan rajapinnan, ja osa rajapinnan jokaisesta määrityksestä on joukko suoritustakuita. Joten esimerkiksi riippumatta siitä, kuinka `vector`-tyyppi on toteutettu, ei pelkästään riitä, että tarjotaan vain saanti sen elementteihin, sen täytyy tarjota *muuttuja-aikainen* saanti. Jos se ei tee niin, se ei ole kelpo vektoritoteutus.

  Dynaamisesti varatut merkkijonot ja taulukot kelpaavat monissa C++-ohjelmissa useimpiin `new`- ja `delete`-funktioiden käyttötapoihin, ja `new/delete`-virheet - varsinkin vuodot, jotka aiheutuvat epäonnistumisesta poistaa `new`-funktiolla varattu muisti - ovat ahdistavan yleisiä. Jos käytät `string`- ja `vector`-olioita (ne molemmat huolehtivat omasta muistinhallinnastaan) `char*`-osoittimien ja osoittimien sijasta dynaamisesti varattuihin taulukoihin, monet `new`- ja `delete`-operaattoreista katoavat, ja niin tapahtuu myös monille vaikeuksille, jotka säännöllisesti liittyvät niiden käyttöön (katso Kohdat 6 ja 11).

- **Algoritmit**. Perussäiliöt ovat mukavia, mutta vielä mukavampaa on, kun on helppo tapa tehdä asioita niillä. Peruskirjasto sisältää yli kaksi tusinaa helppoa tapaa (muun muassa esimääritetyt funktiot, jotka tunnetaan virallisesti nimellä *algoritmit* - ne ovat itse asiasssa malleja), joista useimmat toimivat kirjaston *kaikkien* säiliöiden kanssa - samoin kuin sisäänrakennettujen taulukoiden kanssa!

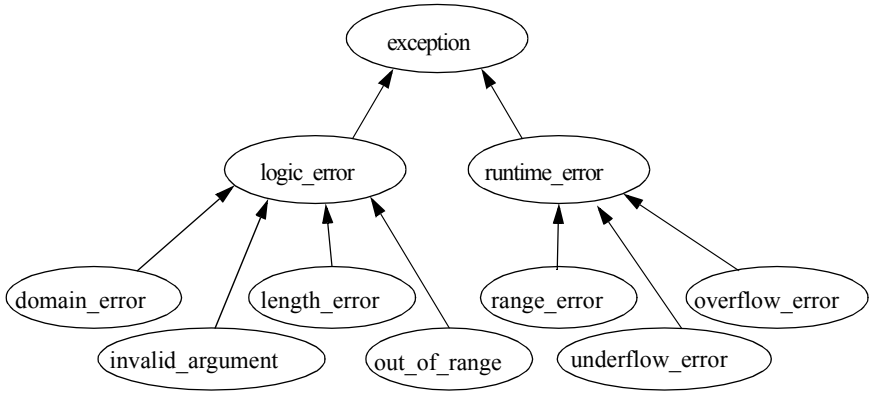
Algoritmit kohtelevat säiliön sisältöä jaksona, ja jokaista algoritmia voidaan käyttää joko jaksoon, joka vastaa säiliön kaikkiin arvoihin, tai alijaksoon. Perusalgoritmien joukossa ovat `for_each` (käytä funktiota jakson jokaiseen elementtiin), `find` (löydä ensimmäinen sijainti jaksossa, jossa säilytetään annettua arvoa), `count_if` (laske jakson niiden elementtien määrä, joissa annettu predikaatti on totta), `equal` (määritä, säilyttävätkö kaksi jaksoa samanarvoisen elementin), `search` (löydä ensimmäinen positio jaksossa, jossa toinen jakso tapahtuu alijaksona), `copy` (kopioi yksi jakso toiseen), `unique` (poista kaksoisarvot jaksosta), `rotate` (pyöräytä jaksossa olevat arvot) ja `sort` (jakson arvojen lajitteluun). Huomaa, että tämä on vain kokoelma saatavilla olevista algoritmeistä; kirjasto sisältää monia muita.

Aivan samoin kuin säiliöoperaatiot toimivat suorituksen takuuna, niin tekevät myös algoritmit. Esimerkiksi, `stable_sort`-algoritmi on pakollinen silloin, kun suoritetaan ei suurempi kuin O (N log N) -vertailuja. (Jos edellisessä lausessa oleva "Big O" -notaatio on vieras sinulle, älä hermostu. Se mitä se todella tarkoittaa on, että leveästi puhuen, `stable_sort`-algoritmin täytyy sisältää suoritus, joka on samalla tasolla kuin kaikkein tehokkaimmat yleiskäyttöiset sarjalajittelun algoritmit.)

- **Tuki kansainvälistymiselle**. Eri kulttuurit tekevät asiat eri tavoilla. Kuten C-kirjasto, C++-kirjastokin tarjoaa piirteitä, joilla helpotetaan kansainvälisten ohjelmistojen tuotantoa, mutta C++-kielen työtapa, vaikka se on käsitteellisesti samaa sukua kuin C-kielen vastaava, on erilainen. Ei pitäisi esimerkiksi olla yllätys sinulle, kun opit, että C++-kielen tuki kansainvälistymiselle hyötyy laajasti malleista, ja se myös hyötyy periytyvyydestä ja virtuaalifunktioista.

  Kirjaston pääasialliset komponentit, jotka tukevat kansainvälistymistä, ovat *fasetit* ja *paikallismuuttujat*. Fasetit kuvaavat, kuinka kulttuurin tietyt erityistuntomerkit pitäisi käsitellä, mukaan lukien numerointisäännöt (eli kuinka paikallisessa merkistössä olevat merkkijonot pitäisi lajitella), kuinka päivämäärät ja ajat ilmaistaan, kuinka numeeriset ja raha-arvot esitetään, kuinka kartoitetaan viestit yksilöijistä (luonnollisiin) kielelle ominaisiin viesteihin, ja niin edelleen. Paikallismuuttujat sitovat yhteen fasetti-joukot. Esimerkiksi Yhdysvaltojen paikallismuuttuja sisältäisi fasetit, jotka kuvaavat, kuinka amerikanenglanniksi lajitellaan merkkijonot, kirjoitetaan päivämäärät ja ajat, luetaan ja kirjoitetaan raha-arvot ja numeeriset arvot ja niin edelleen, tavalla joka on sopivin Yhdysvaltain kansalaisille. Ranskalainen paikallismuuttuja kuvaisi taas, kuinka nämä tehtävät suoritetaan tavalla, johon ranskalaiset ovat tottuneita. C++-kieli sallii useiden paikallismuuttujien olevan aktiivisia yhdessä ohjelmassa, joten ohjelman eri osat voivat soveltaa eri käytäntöjä.

- **Tuki numeeriselle käsittelylle**. FORTRAN-kielen loppu voi lopultakin olla lähellä. C++-kirjasto sisältää kirjaston monimutkaisille numeroluokille (todellisten ja kuvitteellisten osien tarkkuus voi olla `float`, `double` tai `long double`) samoin kuin mallin erityisille taulukkotyypeille, jotka on erityisesti suunniteltu helpottamaan numeerista ohjelmointia. Esimerkiksi oliot, joiden tyyppi on `valarray`, on määritetty tallentamaan elementtejä, jotka ovat vapaita peitenimien käytöstä. Tämä sallii kääntäjien olevan aggressiivisempia optimoinneissaan, varsinkin vektorikoneille. Kirjasto sisältää myös tuen kahdelle erilaiselle taulukkoviipaleelle, samoin kuin algoritmit, joilla lasketaan skalaarit, osasummat, vierekkäiset erot ja paljon muuta.
- **Diagnostinen tuki**. Peruskirjasto sisältää tuen kolmelle tavalle raportoida virheet: C-kielen pakotteiden kautta (katso Kohta 7), virhenumeroiden avulla ja poikkeusten avulla. Jotta poikkeustyypeille saataisiin rakenne, kirjasto määrittää seuraavan hierarkian poikkeusluokille:



Poikkeukset, jotka ovat tyyppiä `logic_error` (tai sen aliluokat), esittävät virheet ohjelman logiikalla. Tällaiset virheet olisi teoriassa voitu estää huolellisemmalla ohjelmoinnilla. Poikkeukset, joiden tyyppi on `runtime_error` (tai siitä periytetyt luokat), esittävät virheet, jotka ovat tunnistettavissa vain suorituksen aikana.

Voit käyttää näitä luokkia sellaisenaan, voit periyttää niistä luoden omat poikkeusluokkasi, tai voit jättää ne huomioimatta. Niiden käyttö ei ole pakollista.

Tämä lista ei kuvaa kaikkea peruskirjastosta löytyvää. Muista, että määritys kattaa yli 300 sivua. Sen pitäisi kuitenkin antaa sinulle peruskäsitys.

Se osa kirjastoa, joka sisältää säiliöt ja algoritmit, tunnetaan yleisesti nimellä *Standard Template Library* (*STL*). On itse asiassa olemassa kolmas komponentti STL-kirjastoon - Iteraattorit - jota en ole vielä kuvannut. Iteraattorit ovat osoitintyyppisiä olioita, joiden avulla STL:n algoritmit ja säiliöt voivat toimia yhdessä. Sinun ei tarvitse ymmärtää iteraattoreita tässä antamassani korkean tason kuvauksessa peruskirjastosta. Jos olet kuitenkin kiinnostunut niistä, voit löytää esimerkkejä niiden käytöstä Kohdasta 39.

STL on kaikkein vallankumouksellisin osa peruskirjastoa, ei pelkästään tarjoamiensa säiliöiden ja algoritmien takia (vaikka ne ovat kieltämättä käytännöllisiä), vaan sen arkkitehtuurin takia. Arkkitehtuuri on yksinkertaisesti sanoen laajennettavissa: STL-kirjastoon voidaan *lisätä*. Peruskirjaston komponentit ovat tietysti kiinteitä, mutta jos noudatat niitä sopimuksia, joiden pohjalle STL on rakennettu, voit kirjoittaa omia säiliöitä, algoritmeja ja iteraattoreita, jotka toimivat aivan yhtä hyvin STL:n peruskomponttien kanssa kuin STL-komponentit toimivat toistensa kanssa. Myös muiden kirjoittamista STL-yhteensopivista säiliöistä, algoritmeista ja iteraattoreista voidaan hyötyä, aivan samoin kuin ne hyötyvät sinun kirjoittamistasi komponenteista. Mikä tekee STL-kirjastosta vallankumouksellisen, on se, että se ei todellakaan ole ohjelma, vaan joukko *sopimuksia*. Peruskirjaston STL-komponentit ovat yksinkertaisesti hyvän ilmentymiä, jotka voivat muodostua noudattamalla näitä sopimuksia.

Kun käytät peruskirjaston komponentteja, voit yleensä ottaen hylätä sen, että suunnittelet omat alusta lähtien rakennetut mekanismit virtauksen syöttöön/tulostukseen, merkkijonoille, säiliöille (mukaan lukien iteraatio ja yleiset manipuloinnit), kansainvälistymiseen, numeerisiin tietorakenteisiin, ja diagnostiikkaan. Saat tämän avulla paljon enemmän aikaa ja energiaa sovelluskehityksen todella tärkeille osille: niiden asioiden toteuttamiseen, jotka erottavat ohjelmasi kilpailijoiden ohjelmista.

Kohta 50: Kehitä C++-kielen ymmärtämystäsi.

C++-kielessä on paljon *kamaa*. C-kielistä kamaa. Kuormittuvaa kamaa. Oliopohjaista kamaa. Mallikamaa. Poikkeuskamaa. Nimiavaruuskamaa. Kamaa, kamaa, kamaa! Joskus tämä kaikki voi olla musertavaa. Kuinka saat järkeä kaikkeen tähän kamaan?

Se ei ole kovin vaikeaa, kunhan ymmärrät työtavan tavoitteet, jotka ovat tehneet C++-kielestä sitä mitä se on. Eturivissä näiden tavoitteiden joukossa ovat seuraavat:

- **Yhteensopivuus C-kielen kanssa**. On olemassa paljon C+-kielellä kirjoitettua, samoin kuin on erittäin paljon C-ohjelmoijia. C++-kieli hyötyy ja rakentuu - tarkoitan, että "vaikuttuu" - tuosta pohjasta.
- **Tehokkuus**. C++-kielen suunnittelija ja ensimmäinen toteuttaja Bjarne Stroustrup tiesi alusta lähtien, että C+-ohjelmoijat, jotka hän toivoi voittavansa puolelleen, eivät katsoisi kahdesti, jos heidän pitäisi maksaa rangaistus suorituksessa vaihdettaessa kieltä. Tästä on tuloksena, että hän varmisti, että C++-kieli oli kilpailukykyinen C-kielen kanssa silloin, kun puhutaan tehokkuudesta - viiden prosentin sisällä.
- **Yhteensopivuus perinteisten työkalujen ja ympäristöjen kanssa**. Hienoja kehitysympäristöjä toimii siellä sun täällä, mutta kääntäjät, linkkerit ja editorit toimivat melkein kaikkialla. C++-kieli on suunniteltu toimimaan ympäristöissä hiiristä mainframe-ympäristöihin, joten sillä on mukanaan mahdollisimman vähän matkatavaroita. Haluatko portata C++-kieleen? Porttaat *kielen* ja hyödyt olemassaolevista työkaluista kohteena olevalla alustalla. (On kuitenkin usein mahdollista tarjota parempi toteutus, jos esimerkiksi linkkeri voidaan muuttaa osoittamaan eräitä avointen funktioiden ja mallien vaativampia näkökulmia.)
- **Soveltuvuus todellisiin ongelmiin**. C++-kieltä ei suunniteltu olemaan mukava, puhdas kieli, joka on hyvä ohjelma opettajille opetettavaksi, se suunniteltiin olemaan voimakas työkalu ohjelmoinnin ammattilaisille, jotka ratkaisevat todellisia ongelmia useanlaisilla toimialueilla. Todellisessa maailmassa on karheat puolensa, joten ei ole yllätys, että tilapäinen ryppy, joka vaikuttaa lopputulokseen käytettäessä työkaluja, joihin ammattilaiset luottavat.

Nämä tavoitteet selittävät joukoittain kielen yksityiskohtia, jotka muuten voisivat pelkästään hiertää. Miksi implisiittisesti luodut kopiomuodostimet ja sijoitusoperaattorit käyttäytyvät omalla tavallaan, varsinkin osoittimiin (katso Kohdat 11 ja 34)? Koska se on tapa, jolla C-kieli kopioi ja sijoittaa `struct`-struktuureita, ja yhteensopivuus C-kielen kanssa on tärkeää. Miksi tuhoajafunktiot eivät ole automaattisesti virtuaalisia (katso Kohta 14), ja miksi toteutusyksityiskohtien pitää olla luokan määrityksissä (katso Kohta 34)? Koska toisin tekeminen saisi aikaan suoritusrangaistuksen, ja tehokkuus on tärkeää. Miksi C++-kieli ei pysty saamaan selville alustusriippuvaisuuksia epäpaikallisten staattisten olioiden välillä (katso Kohta 47)? Koska C++ tukee erillistä kääntämistä (eli kykyä kääntää lähdekieliset moduulit erikseen, ja linkittää useita oliotiedostoja sitten yhteen muodostaakseen suoritettavan tiedoston), luottaa olemassa oleviin linkkereihin, eikä valtuuta ohjelmien tietokantojen olemassaoloa. Tästä on tuloksena, että C++-kielen kääntäjät eivät melkein koskaan tiedä kaikkea koko ohjelmasta. Lopuksi, miksi C++ ei vapauta ohjelmoijia sellaisista uuvuttavista töistä, kuten muistinhallinta (katso Kohdat 5 - 10) ja matalantason osoitinmanipulaatiot? Koska eräät ohjelmoijat tarvitsevat näitä taitoja, ja todellisten ohjelmoijien tarpeet ovat erittäin merkittäviä.

Tämä hädin tuskin vihjaa siitä, kuinka C++-kielen takana olevat suunnittelutavoitteet muokkaavat kielen käyttäytymistä. Kaiken läpikäyminen kattaisi kokonaisen kirjan, joten on mukavaa, että Stroustrup kirjoitti sen. Kirja on *The Design and Evolution of*

C++ (Addison-Wesley, 1994), joka joskus tunnetaan yksinkertaisesti nimellä "D&E". Lue se, niin näet, mitä piirteitä, missä järjestyksessä, ja miksi lisättiin C++-kieleen. Tulet myös oppimaan ne ominaisuudet, jotka hylättiin ja miksi. Voit myös lukea sisäpiirin tietoa siitä, kuinka `dynamic_cast`-ominaisuutta (katso Kohta 39) harkittiin, hylättiin, harkittiin uudelleen ja sitten hyväksyttiin - ja miksi. Jos sinulla on vaikeuksia saada mitään järkevää kuvaa C++-kielestä, D&E:n pitäisi häivyttää paljon tästä hämmennyksestä.

The *Design and Evolution of C++* sisältää paljon oivalluksia siitä, kuinka C++-kielestä muodostui se mikä se on, mutta se on kaikkea muuta kuin muodollinen kielenmääritys. Jos haluat tutustua siihen, sinun täytyy kääntyä C++-kielen kansainvälisen standardin puoleen, joka on vaikuttava muodollisuuksien harjoituskappale sisältäen noin 700 sivua. Voit lukea sieltä tyrmistyttävää proosaa kuten tämä:

> A virtual function call uses the default arguments in the declaration of the virtual function determined by the static type of the pointer or reference denoting the object. An overriding function in a derived class does not acquire default arguments from the function it overrides.
>
> Suomennos edellisestä: Virtuaalifunktiokutsu käyttää static-tyyppisen osoittimen tai olioon viittauksen, määrittämiä oletusargumentteja virtuaalifunktion esittelyssä. Periytetyssä luokassa oleva kumoava funktio ei hanki kumoamansa funktion oletusargumentteja.

Tämä kappale on lähtökohta Kohdalle 38 ("Älä koskaan määritä uudelleen oletuksena olevaa periytynyttä parametrin arvoa"), mutta toivon, että aiheen käsittelyni on jokseenkin helpommin luettavaa kuin edellä mainittu teksti.

Standardi on tuskin mitään unilukemista, mutta se on paras korvauksesi - peruskorvauksesi - jos sinä ja joku muu (sanotaan vaikka kääntäjän myyjä, tai jonkin toisen lähdetekstejä käsittelevän työkalun kehittäjä) olette eri mieltä siitä, mikä on C++-kieltä ja mikä ei ole C++-kieltä. Standardin päätarkoitus on tarjota definitiivistä tietoa, joka ratkaisee tuon kaltaiset argumentit.

Standardin virallinen otsikko täyttää lausuttuna suurenkin suun, mutta jos sinun pitää tietää se, sinun pitää tietää se. Tässä se on: *International Standard for Information Systems - Programming Language C++*. Sen on julkaissut International Organization for Standardization (ISO) -järjestön työryhmä 21 (Working Group 21). (Jos vaatimalla vaadit olla pikkutarkka, sen on todella julkaissut - en ole keksinyt tätä - ISO/IEC JTC1/SC22/WG21.) Voit tilata kopion virallisesta standardista kansainvälisten standardien toimittajalta (Yhdysvalloissa se on ANSI, American National Standards Institute), mutta kopiot standardin viimeisimmistä painoksesta - jotka ovat aika samankaltaisia (vaikka eivät identtisiä) - lopulliseen asiakirjaan ovat vapaasti saatavilla Webistä. Hyvä paikka, josta sitä voidaan etsiä on `http://www.cygnus.com/misc/wp/`, mutta kun ottaa huomioon kyberavaruuden kiihtyvän tahdin, älä hämmästy, jos tämä linkki ei enää toimi, kun yrität sitä. Jos näin on, oma suosikkisi Webin hakukoneista löytää varmasti URL-osoitteen, joka toimii.

Kuten sanoin, *The Design and Evolution of C++* on hieno kirja kielen suunnittelun oivalluksista, ja standardi on upea yhteenveto kielen piirteistä, mutta olisi mukava, jos olisi olemassa välimaasto D&E:n kymmenen kilometrin näkymän ja standardin mikrotason tutkimuksen välillä. Oppikirjojen on tarkoitus täyttää tämä paikka, mutta yleisesti ottaen ne tuntuvat suuntautuvan kohti standardia perspektiiviä, jolloin se, *mitä* kieli on, saa paljon enemmän huomiota kuin se, *miksi* se on mitä se on.

Tervetuloa ARM. ARM on eri kirja, *The Annotated C++ Reference Manual*, kirjoittajina Margaret Ellis ja Bjarne Stroustrup (Addison-Wesley, 1990). Siitä tuli sen julkaisun aikoina auktoriteetti C++-kielessä, ja kansainvälinen standardi käynnistyi ARM:n myötä (mukaanlukien olemassaoleva C-kielen standardi) sen lähtökohtana. Standardin määrittelemä kieli on vuosien varrella eräiltä osin erkaantunut ARM:n kuvaamasta standardista, joten ARM ei enää ole se auktoriteetti mikä se oli. Se on silti käytännöllinen viitemateriaali, koska suurin osa sen asioista pitää edelleen paikkansa, ja on yleistä, että kauppiaat tarttuvat kiinni ARM-määritykseen niissä C++-kielen alueissa, joissa standardi on vasta äskettäin asettautunut.

Se, mikä kuitenkin tekee ARM:sta todella käytännöllisen, ei ole RM-osa (Reference Manual), vaan osa A: huomautukset. ARM sisältää laajan kommentaarin siitä, miksi monet C++-kielen piirteistä käyttäytyvät tavallaan. Osa tästä tiedosta on D&E-kirjassa, mutta paljon sitä ei ole, ja haluat varmasti tietää siitä. Tässä on esimerkiksi jotain, joka tekee useimmat ihmiset hulluiksi, kun he kohtaavat sen ensimmäistä kertaa:

```
class Base {
public:
  virtual void f(int x);
};

class Derived: public Base {
public:
  virtual void f(double *pd);
};

Derived *pd = new Derived;
pd->f(10);                    // virhe!
```

Ongelma on siinä, että `Derived::f`-funktio piilottaa `Base::f`-funktion, vaikka ne ottavat vastaan erilaisia parametrityyppejä, joten kääntäjät vaativat, että `f`-funktioon tapahtuva kutsu ottaa `double*`-osoittimen, mitä literaali 10 ei varmastikaan ole.

Tämä on epämukavaa, mutta ARM sisältää selvityksen tämänkaltaiselle käytökselle. Oletetaan, että kun kutsuit `f`-funktiota, halusit itse asiassa kutsua `Derived`-osassa olevaa versiota, mutta käytit vahingossa väärää parametrityyppiä. Oletetaan edelleen, että `Derived` on kaukana alhaalla periytyvyyden hierarkiassa ja olet epätietoinen, että `Derived` periytyy epäsuorasti samasta kantaluokasta `BaseClass`, ja tuo `BaseClass` esittelee virtuaalifunktion `f`, joka vastaanottaa `int`-tyypin. Tässä tapauksessa olisit huomaamatta kutsunut `BaseClass::f`-funktiota, funktiota

jonka olemassaolosta et edes tiennyt! Tämänkaltainen virhe tapahtuisi säännöllisesti siellä, missä käytetään laajoja luokkahierarkioita, joten Stroustrup päätti tukahduttaa sen alkuunsa niin, että periytetyt luokkajäsenet piilottavat kantaluokan jäsenet nimikohtaisesti.

Huomaa muuten, että jos `Derived`-luokan kirjoittaja haluaa sallia asiakkailleen pääsyn `Base::f`-funktioon, tämä saadaan helposti aikaan käyttämällä yksinkertaisesti esittelyä:

```
class Derived: public Base {
public:
  using Base::f;               // Base::f:n tuonti
                               // Derived-luokan näk.alue.
  virtual void f(double *pd);
};

Derived *pd = new Derived;
pd->f(10);                     // hienoa, kutsuu Base::f
```

Kääntäjille, jotka eivät vielä tue esittelyjä, vaihtoehto on ottaa käyttöön avoimena esitelty funktio:

```
class Derived: public Base {
public:
  virtual void f(int x) { Base::f(x); }
  virtual void f(double *pd);
};

Derived *pd = new Derived;
pd->f(10);                 // toimii, kts Derived::f(int),
                           // joka kutsuu Base::f(int)
```

Saat D&E- ja ARM-kirjojen välissä oivalluksia C++-kielen suunnitteluun ja toteutukseen, joka tekee mahdolliseksi arvostaa hyvin perusteltua, järkevää arkkitehtuuria, joka on joskus barokkimaiselta näyttävän julkisivun takana. Vahvista nämä oivallukset standardin yksityiskohtaisella tiedolla, ja saat perustiedot ohjelmistokehitykseen, joka johtaa C++-kielen todella *tehokkaaseen* käyttöön.