1.4 C++: Moduulit käännösyksiköillä

Seuraavassa luvussa esiteltävä C++:n luokkarakenne sisältää olio-ominaisuuksien lisäksi modulaarisuuden tärkeimmät piirteet: julkisen rajapinnan ja toteutuksen kätkennän. Tässä ja seuraavassa aliluvussa esitellään kaksi tapaa tehdä staattisia moduulirakenteita C++:lla.

Staattisella moduulilla tarkoitetaan käännösaikana luotua ja koko ohjelman suoritusajan samana pysyvää kokonaisuutta, jonka myös kääntäjä ymmärtää erilliseksi yksiköksi. C ja C++ -kääntäjät ovat aina käsitelleet yhtä lähdekooditiedostoa kerrallaan yhtenä kokonaisuutena, jonka käännöksessä pitää olla näkyvillä kaikkien käytettyjen rakenteiden esittelyt. [Kerninghan ja Ritchie, 1988]

Ohjelmoitavan moduulin julkisen rajapinnan esittely voidaan laittaa ns. **otsikkotiedostoon** (*header*, listaus 1.4), joka taas voidaan **#include**-esikääntäjäkomennolla ottaa näkyville kaikkiin moduulia käyttäviin tiedostoihin.

Suurissa ohjelmissa **#include**-rakenteet tulevat mutkikkaiksi ja monitasoisiksi. Kääntäjä voi tällöin virheellisesti saada käsittelyyn saman otsikkotiedoston useaan kertaan saman käännöksen aikana, mikä on virhetilanne koska kieli sallii esittelyiden esiintyvän vain kerran samassa käännöksessä. Esimerkiksi päiväysmoduulin esittelyä tarvitaan useassa korkeamman tason moduulin otsikkotiedostossa, jotka käännösyksikkö ottaa näkyville **#include**-käskyllä, kuten kuva 1.5 seuraavalla sivulla näyttää. Esimerkkilistauksen 1.4 alussa näkyvällä esikääntäjän ehdollisella kääntämisellä (**#ifndef** X **#define** X

```
1  #ifndef PAIVAYS_H
2  #define PAIVAYS_H
3
4  typedef struct paivays_data {
5    int p_, k_, v_;
6  } paivays_PVM;
7
8  paivays_PVM paivays_luo( int paiva, int kuukausi, int vuosi );
9  void paivays_tulosta( paivays_PVM kohde );
         ⋮
10 #endif /* PAIVAYS_H */
```

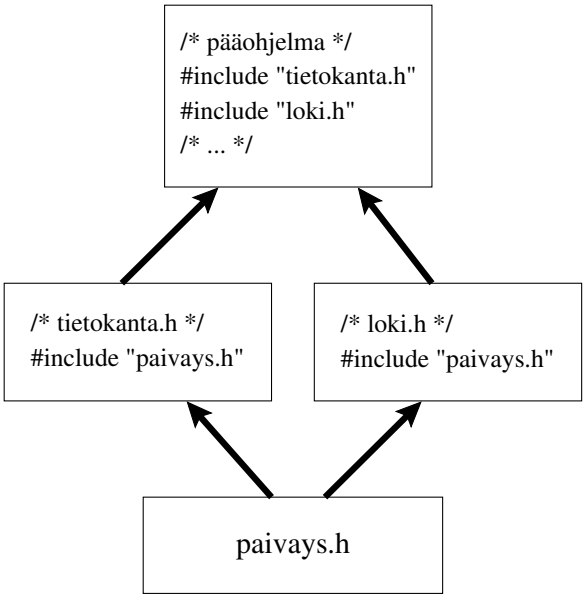
LISTAUS 1.4: Päiväysmoduulin esittely C-kielellä, `paivays.h`

... **#endif**) saadaan varmistettua, että esittely näkyy kääntäjälle vain yhden kerran.

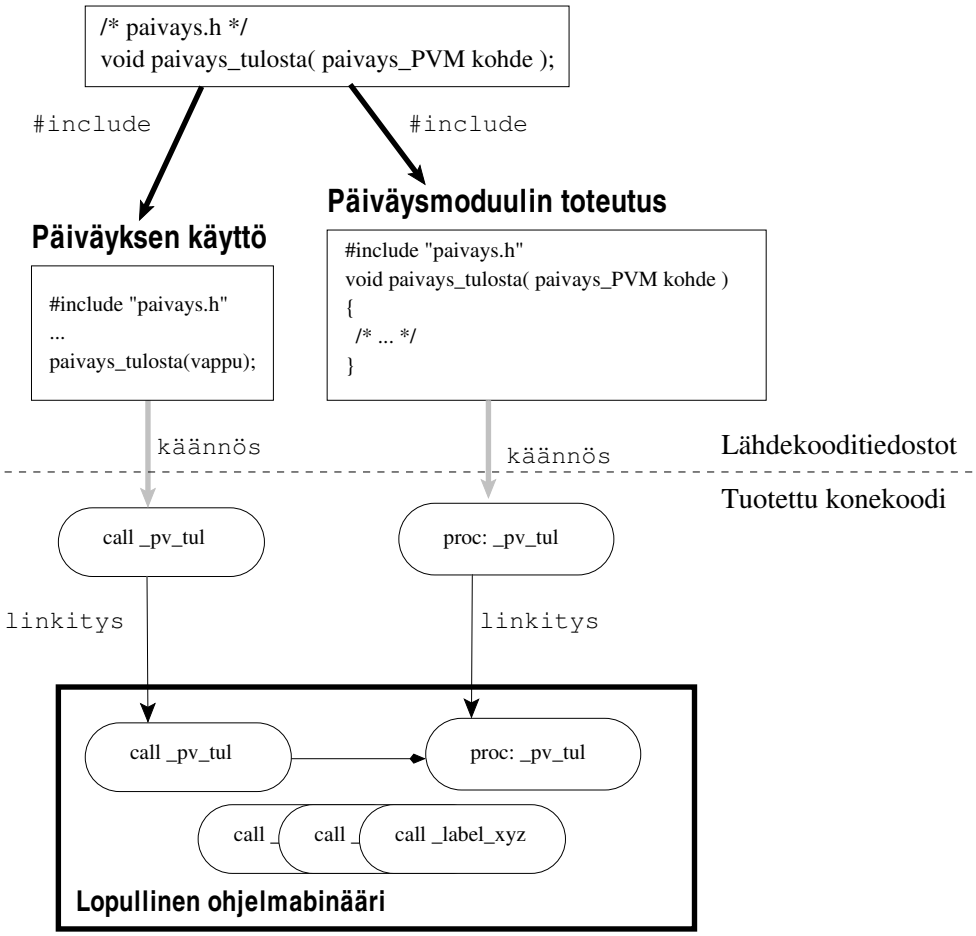
Eri puolella ohjelmistoa (kooditiedostoja) esitellyt rakenteet toteutetaan yhdessä käännösyksikössä, joka lopuksi linkitetään mukaan valmiiseen ohjelmaan. Käännösyksikkö on C-kielessä yksi tiedosto (joka mahdollisesti on ottanut käännökseen mukaan toisia tiedostoja **#include**-rakenteella), josta kääntäjä tekee objektitiedoston. Linkitysvaihe pitää huolen siitä, että esittelyosan nähneiden käännösyksiköiden kutsut moduulin funktioihin menevät oikeaan osaa ohjelmaa lopullisessa suoritettavassa ohjelmabinäärissä (kuva 1.6 seuraavalla sivulla). Tässä ratkaisussa on kaksi ongelmaa:

1. Kaikki eri moduulien käyttämät nimet (funktioiden ja muuttujien nimet) ovat oletuksena käytettävissä kaikkialla ohjelmassa (kunhan ne esitellään käännösyksikössä). Käännösyksikön paikalliseen käyttöön tarkoitetut nimet on ohjelmoijan itse merkittävä määreellä **static**.

2. On olemassa vain yksi ylimmän tason (globaali) näkyvyysalue, jossa kaikki moduulien julkisten rajapintojen symbolit ovat näkyvissä. Jos esim. usea moduuli esittelee funktion `Tulosta`, niin



KUVA 1.5: Sama otsikkotiedosto voi tulla käyttöön useita kertoja



KUVA 1.6: Moduulirakenne C-kielellä

ohjelma ei käänny, koska sama symboli on lopullisessa binäärissä määriteltynä useita kertoja. Tämän nimiavaruuden "roskaantumisen" takia on moduulin toteuttajan pyrittävä nimeämisellä välttämään nimikonflikteja. Esimerkkissä tämä on tehty liittämällä moduulin nimiin etuliite "`paivays_`".

1.5 C++: Nimiavaruudet

Ratkaisuksi suurten ohjelmistojen rajapintojen nimikonflikteihin ja modulaarisen rakenteen esittämiseen ISOC++ -standardi [ISO, 1998] tarjoaa **nimiavaruudet** (*namespace*). Nimiavaruuksien tarkoituksena

on tarjota kielen syntaksin tasolla oleva hierarkkinen nimeämiskäytäntö. Hierarkia auttaa jakamaan ohjelmistoa osiin sekä käytännön ohjelmoinnissa estää nimikonflikteja ohjelmiston eri moduulien välillä. Kuvaavat ja tunnetut nimet voivat esiintyä suuressa ohjelmistossa useassa paikassa ja ne täytyy pystyä erottamaan toisistaan. Esimerkiksi aliohjelmat `Päiväys::tulosta()` ja `Kirjastonkirja::tulosta()` suorittavat saman semanttisen operaation (tulostamisen), mutta etuliite kertoo, minkä moduulin määrittelemästä tulostusoperaatiosta on kyse.

1.5.1 Nimiavaruuksien määritteleminen

Käytännössä C++-ohjelmoija voi uudella avainsanalla **`namespace`** koota yhteen toisiinsa liittyviä ohjelmakokonaisuuksia, jotka voivat olla mitä tahansa C++-ohjelmakoodia. Kaikki esimerkkimme päivämääriä kuvaavat ohjelmiston osat (tietorakenteet, tietotyypit, vakiot, oliot ja funktiot) voidaan kerätä yhteen nimikkeen **`namespace`** `Paivays`[†] alle, kuten listauksessa 1.5 seuraavalla sivulla on tehty.

Nimiavaruuden voi toisessa käännösyksikössä (eli lähdekooditiedostossa) "avata" uudelleen laajentamista varten, jolloin moduulin esittelemät rajapintafunktiot voidaan toteuttaa toisessa ohjelmatiedostossa (listauksessa 1.6 seuraavalla sivulla). Tällä tavoin moduulin esittely ja toteutus saadaan eroteltua toisistaan. Tiedon yksinkertaistamisen (abstrahointi) mukaisesti moduulin käyttäjän pitäisi pystyä hyödyntämään moduulia ainoastaan sen esittelyä (`hh`-tiedosto) ja dokumentointia tutkimalla. "Tylsät" ja epäoleelliset toteutusyksityiskohdat on piilotettu erilliseen käännösyksikköön, joka kuitenkin on osa samaa C++-nimiavaruutta.

1.5.2 Näkyvyystarkenninoperaattori

Kaikki nimiavaruuden sisällä olevat tunnistenimet ovat oletuksena näkyvissä eli käytettävissä vain kyseisen nimiavaruuden sisällä (`paivays.hh`-tiedostossa esitelty tietue `PVM` on käytössä listauksessa 1.6 seuraavalla sivulla). Ulkopuolelta nimiavaruuden sisällä ole-

[†] ISO C++ -standardi sallisi tunnisteiden nimissä mm. skandinaavisia symboleja, jolloin esimerkkimme nimiavaruuden nimi olisi `Päiväys`. Koska käytännössä kääntäjät eivät tällaista ominaisuutta tue, ohjelmaesimerkkiemme nimet ovat ilman suomen kielen kirjaimia å, ä ja ö.

```
1  // Päiväys-moduulin rajapinnan esittely (tiedosto: paivays.hh)
2  #ifndef PAIVAYS_HH
3  #define PAIVAYS_HH
4
5  namespace Paivays {
6
7      // Päiväyksien tietorakenne:
8      struct Pvm { int p_, k_, v_; };
9
10     // Julkisen rajapinnan funktiot:
11
12     Pvm luo( int paiva, int kuukausi, int vuosi );
13           // Palauttaa uuden alustetun päiväyksen.
14
15     void tulosta( Pvm kohde );
16           // Tulostetaan päiväys järjestelmän oletuslaitteelle.
          ⋮
17 }
18 #endif
```

LISTAUS 1.5: Nimiavaruudella toteutettu rajapinta

```
1  // Päiväys-moduulin toteutus (tiedosto: paivays.cc)
2  #include "paivays.hh"        /* rajapinnan esittely */
3
4  namespace Paivays {
5
6      Pvm luo( int paiva, int kuukausi, int vuosi )
7      {
8          Pvm paluuarvo;
          ⋮
9          return paluuarvo;
10     }
11
12     void tulosta( Pvm p )
13     {
          ⋮
14     }
          ⋮
15 }
```

LISTAUS 1.6: Rajapintafunktioiden toteutus erillisessä tiedostossa

viin rakenteisiin voidaan viitata **näkyvyystarkenninoperaattorin** (`::`) avulla (listaus 1.7).

Tarkoituksena on, että ohjelmoija kertoo käyttöpaikassa mitä kokonaisuutta ja mitä alkiota sen sisällä hän kulloinkin tarkoittaa (onko käytössä funktio `Paivays::tulosta()` vai `Kirja::tulosta()`). Tapaa voidaan kritisoida ylimääräisellä kirjoitusvaivalla, mutta tässä kannattaa huomioida myös mukaan tuleva dokumentaatio. Moduulinimet kertovat heti, mitä rakenteita käytetään, eikä niitä luultavasti tarvitse enää erikseen kommentoida ohjelmaan.

Nimiavaruuksia voi olla myös sisäkkäin, jolloin näkyvyystarkentimia ketjutetaan oikean alkion osoittamiseksi (`Tietokanta::Yhteys::SQL`). C++ ei aseta mitään erityisiä rajoja tämän ketjun pituudelle, mutta eivät pitkät ketjut enää noudata nimiavaruuksien alkuperäistä ajatusta selkeydestä hierarkian avulla.

1.5.3 Nimiavaruuksien hyödyt

Toteuttamalla moduulit nimiavaruuksien avulla saadaan C-kielen malliin verrattuna seuraavat parannukset:

- Nimikonfliktien vaara vähenee merkittävästi, koska jokaisen moduulin rajapintanimet ovat omassa nimetyssä näkyvyysalueessaan (tietysti todella laajoissa ohjelmistoissa on pidettävä huoli siitä, etteivät nimiavaruuksien nimet taas vuorostaan ole samoja eri moduuleilla).
- Moduulin määrittelemien rakenteiden käyttö on kielen syntaksin tasolla näkyvän rakenteen (näkyvyystarkennin `::`) vuoksi

```
1 #include "paivays.hh"
2
3 int main() {
4   Paivays::Pvm vappu;
5   vappu = Paivays::luo( 1,5,2001 );
6   Paivays::tulosta( vappu );
        ⋮
7
```

LISTAUS 1.7: Nimiavaruuden rakenteiden käyttäminen

selkeämpää (vertaa esim. Modula-3:n moduuliproseduurien kutsutapa pisteoperaattorilla listauksessa 1.2 sivulla 35).

- Hierarkkisuudesta huolimatta moduulin sisällä on käytettävissä lyhyet nimet. Koodista nähdään syntaksin tasolla esimerkiksi, mitkä funktiokutsut kohdistuvat saman moduulin sisälle ja mitkä muualle ohjelmistoon.

Nimiavaruudet ovat kehittyneet C++:aan vähitellen korjaamaan havaittuja puutteita. Ne ovat melko uusi ominaisuus, joten on olemassa paljon C++ ohjelmakoodia, jossa niitä ei käytetä, mutta nimiavaruuksien etujen takia niiden käyttöä suositellaan yleisesti.

1.5.4 std-nimiavaruus

ISOC++ -standardi määrittelee omaan käyttöönsä `std`-nimisen nimiavaruuden. Tämän nimen alle on kerätty kaikki kielen määrittelyn esittelemien rakenteiden nimet (muutamaa poikkeusta, kuten funktiota `exit`, lukuun ottamatta). Esimerkiksi C:n tulostusrutiini `printf` ja C++:n tulostusolio `cout` ovat ISOC++:n mukaisesti toteutetussa kääntäjässä nimillä `std::printf` ja `std::cout`, jotta ne eivät aiheuttaisi ongelmia ohjelman muiden nimien kanssa. Koska kirjastoon kuuluu myös C-kielestä tulleita funktioita, `std`-nimiavaruus sisältää satoja nimiä. `std`-nimiavaruus on olemassa kaikissa nykyaikaisissa C++-kääntäjissä, ja ainoastaan sen rakenteiden käyttäminen on sallittua — tähän nimiavaruuteen ei saa itse lisätä uusia ohjelmarakenteita.

1.5.5 Standardin uudet otsikkotiedostot

Samalla kun kielen määrittelemät rakenteet on siirretty `std`-nimiavaruuden sisään, ovat myös otsikkotiedostojen nimet vaihtuneet. Aikana ennen nimiavaruuksia C++:ssa otettiin tulostusoperaatiot käyttöön esiprosessorin käskyllä **`#include`** `<iostream.h>`. Nyt oikea (`std`-nimiavaruudessa oleva) tulostusoperaatioiden esittely saadaan käskyllä **`#include`** `<iostream>`. Tämä tiedostotyyppiin viittava `.h`-liitteen puuttuminen on ominaisuus kaikissa standardin määrittelemissä otsikko-“tiedostoissa”. Sana “Tiedostot” on lainausmerkeissä, koska standardi ei määrää rakenteiden olevan missään tiedostossa, vaan em. rivi ainoastaan kertoo kääntäjälle, että kyseiset esittelyt otetaan

käyttöön — kääntäjä saa toteuttaa ominaisuuden vapaasti vaikka tietokantahakuna.

C-kielen kautta standardiin tulleiden otsikkotiedostojen nimistä on myös poistunut loppuliite "`.h`" ja lisäksi niiden eteen on lisätty kirjain "`c`" korostamaan C-kielestä peräisin olevia rakenteita. Esimerkiksi merkkitaulukoiden käsittelyyn tarkoitetut funktiot (`strcpy` yms.) saadaan käyttöön käskyllä **`#include`** <`cstring`>. Näiden "vanhojen" funktioiden käytöstä yhdessä C++-tulostusoperaatioiden kanssa näkyy esimerkki listauksessa 1.8.

1.5.6 Nimiavaruuden synonyymi

Nimiavaruuden nimi voi myös olla synonyymi (alias) toiselle jo olemassa olevalle nimiavaruudelle. Tällöin kaikki alias-nimeen tehdyt viittaukset käyttäytyvät alkuperäisen nimen tavoin. Tätä ominaisuutta voidaan hyödyntää esimerkiksi silloin, jos samalle moduulille on olemassa useita vaihtoehtoisia toteutuksia. Moduulia käyttävä ohjelmakoodi kirjoitetaan käyttämällä alias-nimeä ja todellinen käyttöön otettava moduuli valitaan synonyymin määrittelyn yhteydessä (katso listaus 1.9 seuraavalla sivulla).

Kun ollaan nimeämässä yleiskäyttöisiä ohjelmakoodikirjastoja, on tärkeätä valita myös kokonaisuuden nimiavaruudelle nimi, joka ei helposti aiheuta nimikonfliktia muun ohjelmiston kanssa. Esimer-

```
1  #include <cstdlib>       // pääohjelman paluuarvo EXIT_SUCCESS
2  #include <iostream>      // C++:n tulostus
3  #include <cstring>       // C:n merkkitaulukkofunktiot
4
5  int main()
6  {
7     char const* const p = "Jyrki Jokinen";
8     char puskuri[ 42 ];
9     std::strcpy( puskuri, "Jyke " );
10    std::strcat( puskuri, std::strstr(p, "Jokinen") );
11
12    std::cout << puskuri << std::endl;
13    return EXIT_SUCCESS;
14 }
```

LISTAUS 1.8: C-kirjastofunktiot C++:n nimiavaruudessa

```
1  #include "prjlib/string.hh"
2  #include <string>
3
4  int main() {
5  #ifdef PRJLIB_OPTIMOINNIT_KAYTOSSA
6    namespace Str = ComAcmeFastPrjlib;
7  #else
8    namespace Str = std;
9  #endif
10
11   Str::string esimerkkijono;
         ⋮
12 }
```

LISTAUS 1.9: Nimiavaruuden synonyymi (aliasointi)

kiksi `String` lienee huono kirjaston nimi, sillä usealla valmistajalla on varmasti kiinnostusta tehdä samanlainen rakenne. Virallisen nimen kannattaa olla pitkä (`OhjTutFiOpetusMerkkijono`), ja ohjelmoijat voivat omassa koodissaan käyttää moduulia lyhyemmällä etuliitteellä synonyymin avulla.

1.5.7 Lyhyiden nimien käyttö (`using`)

Etuliitteen "`std::`" toistaminen jatkuvasti esimerkiksi `cout`-tulostuksessa on varmasti raskaalta ja turhalta tuntuva rakenne.[8] Jos samassa ohjelmalohkossa käytetään useita kertoja samaa nimiavaruuden sisällä olevaa nimeä, ohjelmoijien kirjoitusvaivaa helpottamaan on olemassa **`using`**-lause, joka "nostaa" nimiavaruuden sisällä olevan nimen käytettäväksi ohjelman nykyisen näkyvyysalueen sisälle. Peruskäytössä kerrotaan yksittäinen rakenne, jota halutaan käyttää. Esimerkiksi **`using`** `std::cout` mahdollistaa tulostusolion nimen käytön ilman etuliitettä (katso listaus 1.10 seuraavalla sivulla). Jotta nimiavaruuksien alkuperäinen käyttötarkoitus säilyisi, **`using`**-lauseet tulisi laittaa aina mahdollisimman lähelle niiden tarvittua käyttökohtaa ohjelmatiedostossa (funktion tai koodilohkon alkuun, jossa ne samalla toimivat dokumenttia käytetyistä ulkopuolisista rakenteista).

[8] Kaikki C++-kääntäjät eivät vaadi `std::`-etuliitteen käyttämistä, mutta kyseessä on kielen standardin vaatimuksen vastainen toiminnallisuus.

```
1  #include "paivays.hh"
2  #include <iostream>
3
4  void kerroPaivays( Paivays::Pvm p )
5  {
6     using std::cout; // käytetään määrättyä nimeä
7     using std::endl;
8     using namespace Paivays; // käytetään kaikkia nimiavaruuden nimiä
9     cout << "Tänään on: ";
10    tulosta( p ); // Kutsuu rutiinia Paivays::Tulosta
11    cout << endl;
12 }
```

LISTAUS 1.10: `using`-lauseen käyttö

Turvallisuussyistä **`using`** tulisi laittaa ohjelmakooditiedoston alkuun vasta kaikkien `#include`-käskyjen jälkeen, jottei se vahingossa sotke otsikkotiedostojen toimintaa väärään paikkaan nostetulla nimellä. Sama sotkemisen välttäminen tarkoittaa myös sitä, ettei **`using`**-lauseita kannata laittaa otsikkotiedostojen sisälle (etukäteen ei tiedetä missä yhteydessä tai järjestyksessä tiedoston sisältämiä esittelyitä käytetään). [Sutter, 2000]

Hyvä perusperiaate on käyttää **`using`**-lausetta mahdollisimman lähellä sitä aluetta, jossa sen on tarkoitus olla voimassa (yleensä koodilohko tai funktio). Toisaalta paljon käytetyissä rakenteissa on usein dokumentaation kannalta selkeämpää kirjoittaa **`using`** heti siihen liittyvän otsikkotiedoston `#include`-käskyn jälkeen. Tämä suositus taas on heti ristiriidassa edellisen kappaleen “sotkemissäännön” kanssa. Nimiavaruudet ovat C++:ssa sen verran uusi asia, että parasta mahdollista suositusta selkeyden ja turvallisuuden kannalta tässä asiassa on vaikea antaa. Hyvä kompromissi on luottaa kääntäjän `std`-kirjastojen olevan oikein toimivia, jolloin niiden esittelyiden väliin voi kirjoittaa huoletta **`using`**-lauseita, mutta omissa osissa ja ostettujen kirjastojen kanssa kirjoittaa **`using`**-lauseet vasta kaikkien esittelyiden (otsikkotiedostojen) jälkeen. Listaus 1.11 seuraavalla sivulla on esimerkki tästä “yhdistelmäsäännöstä”.

Kun normaali **`using`**-lause nostaa näkyville yhden nimen, sen erikoistapaus **`using namespace`** nostaa nimiavaruuden sisältä ***kaikki*** nimet nykyiselle tasolle ja siten käytännössä poistaa nimiavaruuden käytöstä kokonaan. Erityisesti lausetta **`using namespace std`** tulisi ai-

```
1  // Omat rakenteet esitellään std-kirjastoja ennen
2  // (tämä siksi että saamme tarkastettua niiden sisältävän kaikki
3  // tarvittavat #include-käskyt ts. ne ovat itsenäisesti kääntyviä yksikköjä)
4  #include "paivays.hh"
5  #include "swbus.hh"
6  #include "tietokanta.hh"
7  #include "loki.hh"
8  // Kaikista yleisimmin tässä tiedostossa käytetyt std-rakenteet esitellään
9  // heti niihin liittyvän otsikkotiedoston jälkeen
10 #include <iostream>
11 using std::cout;
12 using std::endl;
13 #include <vector>
14 using std::vector;
15 #include <string>
16 using std::string;
17 // lopuksi omiin moduuleihin liittyvät using-lauseet
18 using Paivays::PVM;
19 using Loki::varoitus;
20 using Loki::virhe;
```

LISTAUS 1.11: `using`-lauseen käyttö eri otsikkotiedostojen kanssa

na välttää C++-ohjelmissa. Tätä rakennetta käytetään kyllä usein opetuksessa ja lyhyissä esimerkkiohjelmissa selkeyden saavuttamiseksi, mutta suurissa C++-ohjelmistoissa emme sitä suosittele.

1.5.8 Nimeämätön nimiavaruus

Nimiavaruuksiin on määritelty erikoistapaus **nimeämätön nimiavaruus** (*unnamed namespace*), jolla rajoitetaan funktioiden ja muuttujien nimien näkyvyys yhteen käännösyksikköön (tämä tehtiin aikaisemmin C- ja C++-kielissä **`static`**-avainsanan avulla). Nimeämättömällä nimiavaruudella voidaan dokumentoida ne osat moduulin ohjelmakoodia, jotka ovat olemassa ainoastaan sen sisäistä toteutusta varten. Samalla kääntäjä myös pitää huolen, ettei niitä vahingossa pystytä käsittelemään moduulin ulkopuolelta.

Listauksessa 1.12 seuraavalla sivulla määritellään yksi muuttuja ja funktio käännösyksikön paikallisiksi nimeämättömällä nimiavaruudella. Tämän nimiavaruuden sisällä olevat rakenteet ovat samassa käännösyksikössä (tiedostossa) suoraan käytettävissä ilman näkyvyystarkenninta (jota nimeämättömällä nimiavaruudella ei tietysti

edes ole), mutta ne eivät ole käytettävissä käännösyksikön ulkopuolella.

Nimeämätön nimiavaruus suojaa sen sisällä olevat nimet tavallisen nimiavaruuden tapaan. Vaikka useassa käännösyksikössä on samoilla nimillä olevia rakenteita, niin ne eivät sotke toisiaan, kunhan kaikki ovat nimeämättömän nimiavaruuden sisällä. Tämän voi ajatella tapahtuvan siten, että kääntäjä tuottaa kulissien takana uniikin nimen jokaiselle nimeämättömälle nimiavaruudelle. Jos nimeämätöntä nimiavaruutta käyttää tavallisen nimiavaruuden sisällä, on nimeämättömän nimiavaruuden sisältö käytettävissä ainoastaan tämän tavallisen nimiavaruuden sisällä (esim. moduulin sisäinen funktio tai tietorakenne).

```
1  static unsigned long int laskuri; // Vanha tapa
2
3  // ISO C++:n mukainen tapa
4  // nimeämätön nimiavaruus:
5  namespace {
6     unsigned long int viiteLaskuri;
7     void lisaaViiteLaskuria() {
8        ++viiteLaskuri;
            ⋮
9     }
10 }
11
12 // Julkinen operaatio:
13 Pvm luoPaivaysOlio() {
14    lisaaViiteLaskuria();
15    // ylläoleva rutiinin kutsu toimii samassa tiedostossa ilman
16    // using-lausetta tai näkyvyystarkenninta.
17    // rutiinia ei pysty kutsumaan tiedoston ulkopuolisesta koodista.
            ⋮
18 }
19
```

LISTAUS 1.12: Nimeämätön nimiavaruus