C++-tyyliopas

Tampereen teknillinen yliopisto,
Ohjelmistotekniikka, 2005

Opettele säännöt, jotta tiedät miten rikkoa niitä oikein.

– Tuntematon

Tämä C++-tyyliopas on Tampereen teknillisen yliopiston Ohjelmistotekniikan laitoksen opetuskäyttöön tarkoitettu malli. Oppaaseen kerätyt suositukset löytyvät *Matti Rintalan* ja *Jyke Jokisen* kirjasta "Olioiden ohjelmointi C++:lla" [Rintala ja Jokinen, 2005]. Tässä erillisessä vapaasti levitettävässä versiossa on ohjeisiin lisätty selittäviä kommentteja. Oppaan tuorein versio on saatavilla WWW-osoitteessa `http://www.cs.tut.fi/~oliot/kirja/tyyliopas/`.

Koska nykyaikaiset ohjelmointikielet ovat mutkikkaita ja monitahoisia työkaluja, tyylioppaalla pyritään rajaamaan projektin tai organisaation tarpeisiin sopivat tavat käyttää tätä työkalua. Yhtenäisen ohjelmointityylin noudattaminen lisää lähdekoodin ymmärrettävyyttä, ylläpidettävyyttä, siirrettävyyttä, luettavuutta ja uudelleenkäytettävyyttä. Mutkikkaissa ohjelmointikielissä (kuten C++) tyylioppaan ohjeet voivat myös lisätä koodin tehokkuutta ja auttaa välttämään ohjelmointivirheitä — tai ainakin auttaa löytämään virheet nopeammin. Hyvä tyyliopas pystyy parantamaan kirjoittamamme ohjelmakoodin laatua. Tähän tyylioppaaseen on koottu sääntöjä, joita noudattamalla pääsee kohti tuota päämäärää käytettäessä C++-ohjelmointikieltä.

Opas on kokoelma sääntöjä ja suosituksia. Säännöt ovat perusteltavissa ISO C++ -standardilla tai muilla erityisen painavilla syillä. Ohjelmakoodin ulkoasuun, muokkaamiseen ja taltiointiin liittyvät suositukset on tarkoitettu ***yhdenmukaisiksi sopimuksiksi***, joita tarvittaessa muokataan tyyliohjetta sovellettaessa. Suositus "tiedostonimi päättyy aina päätteeseen `.cc`" voidaan tarvittaessa muokata tarkoittamaan esimerkiksi tiedostopäätettä `.C`, `.cxx` tai `.cpp` käytety käännösympäristön vaatimusten mukaisesti.

1. Yleistä

1.1. Kaikkia tyylioppaan sääntöjä on noudatettava.

1.2. Tyylioppaan säännöstä saa aina poiketa hyvällä perusteella, joka tulee kirjata ohjelmakoodin kommentteihin.

1.3. Säännöt eivät koske ulkopuolista lähdekoodia (muualta hankittu tai koodigeneraattorin tuottama).

1.4. Vanhoja ohjelmia ei muuteta tämän tyylioppaan mukaisiksi, vaan niitä ylläpidetään niitä tehtäessä noudatetun käytännön mukaisesti.

2. Ohjelmiston tiedostot

2.1. Tiedostot ovat määrämuotoisia: alussa on koko tiedostoa koskeva kommentti.

2.2. Tiedoston kommentit kirjoitetaan C++-muodossa (`// ...`). Monen rivin kommentit voidaan sijoittaa C-kielen kommenttien (`/* ... */`) sisään.

2.3. Otsikkotiedostot

2.3.1. Otsikkotiedostojen nimen pääte on `.hh`

> *Yhtenäiseen nimeämiseen kuuluvat määrämuotoiset tiedostonimet. Päätteen valintaan vaikuttaa ensisijaisesti se, mitä tiedostoja käytetty käännösympäristö pitää oletusarvoisesti C++-tiedostoina. Päätettä `.h` ei kannata käyttää, koska se on jo käytössä C-kielisissä ohjelmissa.*

2.3.2. Yhdessä otsikkotiedostossa esitellään yksi julkinen rajapinta (luokka tai nimiavaruus).

> *Hyvin dokumentoitu julkinen rajapinta on usein sopivan kokoinen kokonaisuus sijoitettavaksi yhteen ohjelmakooditiedostoon. (Katso myös kohta 5.1.)*

2.3.3. Otsikkotiedosto on selkeästi kommentoitu, jos sen perusteella pystytään suunnittelemaan ja toteuttamaan rajapintaa koskevat testitapaukset.

2.3.4. Otsikkotiedostoissa on ainoastaan esittelyitä. (ISO C++:n ominaisuuksien takia otsikkotiedostossa voi olla myös kokonaislukuvakioita, **`inline`**-rakenteita [Rintala ja Jokinen, 2005, aliluku A.2] ja **`template`**-malleja [aliluku 9.5].)

> *Vaikka **`inline`**- ja **`template`**-rakenteet sisältävät toteutuksen ohjelmakoodia, niin nykyisten C++-kääntäjien on nähtävä niiden täysi määrittely, ennen kuin kyseisiä rakenteita voidaan käyttää (instantioida) muussa ohjelmakoodissa.*

2.3.5. Laajat mallit ja **`inline`**-koodit tulee kirjoittaa erilliseen tiedostoon (jolla on yhtenäinen pääte esimerkiksi `.icc`). Tämä tiedosto luetaan otsikkotiedoston lopussa `#include`-käskyllä.

Toteutustapa jolla sääntöä 2.3.4 voidaan noudattaa selkeästi.

2.3.6. Otsikkotiedostossa otetaan käyttöön (**`#include`**-käskyillä) ainoastaan ne muut esittelyt, jotka ovat tarpeen tiedoston sisältämän esittelyn takia.

Ylimääräiset `#include`-käskyt aiheuttavat turhia riippuvuuksia lähdekooditiedostojen välille vaikeuttaen ylläpidettävyyttä.

2.3.7. Otsikkotiedostoissa ei saa käyttää **`using`**-lauseita. [Rintala ja Jokinen, 2005, aliluku 1.5.7] (Katso myös kohta 3.3.)

Esittelyiden yhteydessä on suurin vaara nimikonflikteista eri otsikkotiedostojen sisältämien nimien välillä. Eksplisiittinen nimien käyttö näkyvyystarkentimella ei sotke ohjelman nimiavaruuksia tai tuota ohjelmakoodia, jonka toimivuus riippuu käytettyjen otsikkotiedostojen keskinäisestä käyttöönottojärjestyksestä.

2.3.8. Jos esittelyssä (yleensä otsikkotiedostossa) viitataan johonkin luokkatyyppiin vain osoittimella tai viitteellä, käytetään ennakkoesittelyä [Rintala ja Jokinen, 2005, aliluku 4.4]. ***Ennakkoesittelyn saa tehdä vain itse tekemilleen luokille***.

*Vain itse tehdystä ohjelmakoodista voi varmasti tietää ennakkoesittelyn olevan sallittua. Esimerkiksi vaikka standardikirjaston `std::string` [Rintala ja Jokinen, 2005, aliluku A.4] vaikuttaa luokkatyypiltä, siihen ei saa tehdä ennakkoesittelyä, koska toteutus on tyyppialias (**`typedef`**).*

2.3.9. Otsikkotiedosto on suojattava moninkertaiselta käyttöönotolta [Rintala ja Jokinen, 2005, aliluku 1.4].

`#include`-ketjuissa moninkertainen otsikkotiedoston käyttöönotto on mahdollista ja aiheuttaa käännösvirheen (C++-rakenteet voidaan esitellä kääntäjälle vain kerran saman käännöksen yhteydessä). Suojaus toteutetaan ehdollisen kääntämisen esiprosessorikäskyillä otsikkotiedoston alussa ja lopussa:

```
#ifndef LUOKKA_A_HH
#define LUOKKA_A_HH
      ⋮
#endif // LUOKKA_A_HH
```

2.3.10. `#include`-käskyissä ei saa esiintyä viittauksia tietyn koneen tiettyyn hakemistoon.

Kaikki yhteen käännösympäristöön sitovat viittaukset vaikeuttavat lähdekoodin ylläpitoa ja siirrettävyyttä.

2.4. Toteutustiedostot

2.4.1. Toteutukset sisältävän tiedoston pääte on `.cc`

2.4.2. Toteutustiedostossa otetaan käyttöön vain kyseisen käännösyksikön toteuttamisen kannalta tarpeelliset otsikkotiedostot.

2.4.3. Toteutustiedostossa otetaan ensin käyttöön ohjelman paikalliset esittelyt (otsikkotiedostot) ja vasta sitten käännösympäristön ja ISO C++-kielen määrittelemät kirjastot [Rintala ja Jokinen, 2005, aliluku 1.5.7] (katso myös kohta 3.3.5).

Ulkopuolisten esittelyiden keskinäisestä järjestyksestä riippuvat otsikkotiedostot ovat vaarallista ohjelmakoodia ja tällä käytännöllä ne huomataan helpommin käännöksen yhteydessä.

2.4.4. Rajapinnan toteutukset esitetään tiedostossa samassa järjestyksessä kuin ne ovat vastaavassa otsikkotiedostossa.

Näin määrätyn rajapintaoperaation toteutuksen löytäminen helpottuu.

3. **Nimeäminen**

3.1. Ohjelmakoodin kommentit ja symboliset nimet kirjoitetaan yhtenäisellä kielellä ja nimeämistavalla.

Esimerkiksi TTY:n opetuskieli on suomi, joten on loogista käyttää samaa kieltä myös ohjelmoidessa. Vastaavasti jos yrityksen virallinen kieli on englanti, se lienee parempi vaihtoehto ohjelmakoodin nimien ja kommentoinnin kieleksi.

3.2. Alaviivalla alkavia nimiä ei saa käyttää. Samoin kiellettyjä ovat nimet, joissa on kaksi peräkkäistä alaviivaa. [Rintala ja Jokinen, 2005, aliluku 2.3.2].

ISO C++ -standardi varaa kääntäjän ja kirjastojen toteutusten käyttöön tietyt alaviivalla alkavat nimet sekä kaikki kaksi peräkkäistä alaviivaa sisältävät nimet.

3.3. Nimiavaruudet

3.3.1. Moduulin käyttämät nimet pidetään erillään muusta ohjelmistosta sijoittamalla ne nimiavaruuden (**`namespace`**) sisään [Rintala ja Jokinen, 2005, aliluku 1.5.1].

Moduulin sisältämille rakenteille saadaan nimiavaruuden nimestä selkeä yhtenäinen etuliite. Samalla vähennetään merkittävästi nimikonfliktien riskiä ohjelmiston eri osien välillä.

3.3.2. ISO C++:n uusia otsikkotiedostoja ja niiden kautta std-nimiavaruutta on käytettävä. [Rintala ja Jokinen, 2005, aliluku 1.5.4] ja [aliluku 1.5.5].

Standardin otsikkonimissä ei ole olemassa tiedostopäätettä ja kaikki C-kielestä tulevat otsikkotiedostot alkavat kirjaimella "c".

```
#include <cstring>
#include <iostream>
using std::cout;
using std::endl;

void tulostaVirhe( char* m )
{
  std::cerr << std::strstr( m, "err" ) << endl;
}
```

3.3.3. Jos nimiavaruuden sisältä nostetaan näkyville nimiä, niin käyttöön otetaan vain todella tarvittavat nimet (lauseen **using namespace** X käyttö on kiellettyä). [Rintala ja Jokinen, 2005, aliluku 1.5.7]

3.3.4. **using**-lauseella nostetaan nimet käyttöön lähelle käyttöpaikkaa (koodilohko tai funktio).

3.3.5. Erityisen usein tarvittavat nimet voidaan nostaa näkyville koko käännösyksikköön ja niitä koskevat **using**-lauseet tulee sijoittaa tiedoston kaikkien #include-käskyjen jälkeen, jotta "nostetut" nimet eivät pääse sotkemaan toisia esittelyitä. ISO C++:n otsikkotiedostoja voidaan kuitenkin pitää hyvin muotoiltuina ja niiden yhteyteen liitetyt kyseistä esittelyä koskevat **using**-lauseet usein parantavat ohjelmakoodin luettavuutta. [Rintala ja Jokinen, 2005, aliluku 1.5.7]

```
#include "prjlib.hh"
#include <iostream>
using std::cout;
using std::endl;
#include <sstream>
using std::ostringstream;

using Prjlib::Puskuri;
using Prjlib::Paivays;
```

3.3.6. Käännösyksikön (tiedosto) sisäisessä käytössä olevat määrittelyt sijoitetaan nimeämättömän nimiavaruuden sisään. [Rintala ja Jokinen, 2005, aliluku 1.5.8]

Oletuksena käännösyksikön nimet voivat aiheuttaa nimikonflikteja muun ohjelmiston osien kanssa linkitysvaiheessa. Nimeämätön nimiavaruus varmistaa määrittelyjen olevan uniikkeja (käännösyksikön sisäisiä).

```
namespace { // nimeämätön nimiavaruus
  unsigned long int viiteLaskuri = 0;
  void lisaaViiteLaskuria() {
      ⋮
  }
}
```

3.4. Luokka- (**`class`**) ja tietuetyyppien (**`struct`**), nimiavaruuksien (**`namespace`**) sekä tyyppialiaksien (**`typedef`**) nimet alkavat isolla alkukirjaimella.

3.5. Muuttujien, funktioiden ja jäsenfunktioiden nimet alkavat pienellä alkukirjaimella. Useammasta sanasta koostuvat nimet kirjoitetaan yhteen ja loppupään sanat aloitetaan isolla alkukirjaimella (esimerkiksi: `haeRaportinPaivays()`). ***Poikkeus sääntöön on luokan rakentaja, jonka nimen on oltava täsmälleen sama kuin luokan nimi***.

3.6. Jäsenmuuttujien nimen viimeinen merkki on alaviiva (tai ne erotellaan muista nimistä jollain muulla yhtenäisellä nimeämistavalla) [Rintala ja Jokinen, 2005, aliluku 2.3.2].

Jäsenmuuttujien yhtenäinen nimeäminen helpottaa niiden tunnistamista luettaessa jäsenfunktioiden toteutuskoodia. (Toinen usein käytetty nimeäminen on liittää kaikkien jäsenmuuttujien nimen eteen englannin omistamista tarkoittava sana "my".)

3.7. Luokkamuuttujien nimen viimeinen merkki on alaviiva (tai ne erotellaan muista nimistä ja mahdollisesti myös jäsenmuuttujista jollain muulla yhtenäisellä tavalla) [Rintala ja Jokinen, 2005, aliluku 8.2.2].

Kohdan 3.6 mukainen nimeäminen, joka helpottaa luokkamuuttujien tunnistamista ohjelmakoodissa.

3.8. Vakiot kirjoitetaan kokonaan isoilla kirjaimilla ja sanojen erottimena käytetään tarvittaessa alaviivaa. ("Vakion" arvo on aina sama. Jos esim. funktion paikallisen const-muuttujan arvo lasketaan ajoaikana ja voi vaihdella funktion kutsukerroilla, kannattaa se varmaan nimetä tavallisen muuttujan tapaan.)

```
unsigned int const PUSKURIN_SUURIN_KOKO = 1024;
```

4. Muuttujat ja vakiot

4.1. Muuttujien näkyvyysalue tulee suunnitella mahdollisimman pieneksi (koodilohko, funktio, luokka tai nimiavaruus).

Lähellä käyttöpaikkaa määritellyt ja alustetut muuttujat lisäävät ohjelmakoodin selkeyttä.

4.2. Jokainen muuttuja on määriteltävä eri lauseessa.

Muuttujan tyyppi, nimi, alustus ja sitä koskeva kommentti ovat usein yhdelle riville sopiva kokonaisuus. Sääntö myös varmistaa, että kohdan 5.4 mukaiset osoittimien ja viitteiden määrittelyt tulevat aina oikein. "`char* p1, p2;`" olisi luultavasti virhe, sillä nyt `p1` on osoitin ja `p2` merkkimuuttuja.

4.3. Ohjelmassa ei saa olla alustamattomia muuttujia.

Jos muuttujalla ei ole järkevää alkuarvoa, sille annetaan jokin "laiton" alkuarvo (nolla, miinus yksi, tms.). Alustamattomat muuttujat ovat yksi suurimmista (satunnaisten) virhetoimintojen aiheuttajista ohjelmistoissa. Laittomaan arvoon alustetut muuttujat eivät korjaa virheitä, mutta tuovat ne testeissä helpommin esille.

4.4. Luokan jäsenmuuttujat alustetaan aina rakentajan alustuslistassa, jossa jäsenmuuttujat luetellaan niiden esittelyjärjestyksessä [Rintala ja Jokinen, 2005, aliluku 3.4.1].

Järjestyksen määrää ISO C++ -standardi. Alustuslista on myös ainoa tapa alustaa jäsenmuuttujana olevat oliot tehokkaasti (kertomalla niiden alustukseen käytettävä rakentaja).

4.5. **`#define`**-käskyä ei saa käyttää ohjelman vakioiden ja "makrofunktioiden" määrittelyyn [Rintala ja Jokinen, 2005, aliluku 4.3].

*Nimetyt vakiot toteutetaan C++:ssa **`const`**-muuttujina ja makrot korvataan **`inline`**-funktioilla.*

4.6. Ohjelmakoodin toimintaa ohjaavat raja-arvot ja muut vakiot määritellään keskitetysti yhdessä paikassa vakiomuuttujiksi (numeeristen vakioiden käyttö ohjelmassa on kiellettyä). Lyhyet merkitykseltään varmasti pysyvät vakiot kuten esimerkiksi nollaosoitin (`0`) tai "kymmenen alkion silmukka" (`-5 ... 5`) on kuitenkin usein selkeämpää kirjoittaa suoraan käyttöpaikassa näkyviin.

Kaikki numeeriset arvot tulisi määritellä yhdessä keskitetyssä paikassa (otsikkotiedosto) selkeästi nimetyiksi vakiomuuttujiksi, joita käytetään muualla ohjelmakoodissa. Vakioiden arvojen muuttaminen myöhemmin on tällöin helppoa. Näin tulisi tehdä kaikille arvoille, joita mahdollisesti halutaan muuttaa ohjelman jatkokehityksessä. Jos tiedetään hyvin varmaksi, ettei jokin arvo muutu (nolla tai "tässä laiteohjaimessa on kymmenen alkion alustus"), niin arvon kirjoittaminen ilman vakiomuuttujan tuomaa epäsuoruutta on perusteltua.

4.7. Totuusarvot ilmaistaan ISO C++:n tietotyypillä **bool**, jonka mahdolliset arvot ovat **false** ja **true**.

C-kielen "totuusarvot" nolla ja nollasta poikkeavat kokonaisluvut ovat edelleen toimivia, mutta niiden käyttöä ei suositella ISO C++:ssa.

4.8. Kun tyyppi määritellään vakioksi, **const**-sanan paikka tyypin määrittelyssä on itse tyyppinimen *jälkeen*. Tämä suositeltava tyyli on yleistymässä uusissa ohjelmissa. Aiemmin koodissa on ollut tapana kirjoittaa **const** *ennen* tyyppinimeä. Tärkeintä on, että käytäntö on yhtenäinen koko ohjelmassa. [Rintala ja Jokinen, 2005, aliluku 4.3]

```
int const VAKIO = 3;       // Kokonaislukuvakio
int const* ptr = &VAKIO; // vakio-osoitin eli osoitin vakioon
int* const PTR2 = 0;       // Osoitinvakio jota ei saa muuttaa
```

*Tapa laittaa **const**-sana vasta tyypin nimen jälkeen on C++:n kannalta loogisempi, ja sitä näkee käytettävän yhä enemmän uudessa C++-koodissa. Vastaavat muutkin tyypin määreet kuten osoitin-* ja viite-& tulevat vasta tyypin jälkeen. Tällä on merkitystä kun määreitä on monta peräkkäin. Erityisesti uusi tapa on selvempi template-koodia kirjoitettaessa:*

```
template <typename T>
T const f(T p)
{
  return p;
}

      ⋮

int* p = 0;
int* const p2 = f(p);
// Paluuarvo on tässä tapauksessa int* const, ei const int*
```

5. Asemointi ja tyyli

5.1. Yli tuhannen rivin tiedostojen käsittely on hankalaa, joten niitä tulisi välttää.

5.2. Yhden (jäsen)funktion toteutuksen pituuden ei tulisi ylittää sataa riviä.

5.3. Päätelaitteiden käytännöksi on muodostunut 80 merkkiä leveä näyttö. Tästä johtuen yli 79 merkin pituiset rivit kannattaa jakaa useammalle riville.

```
std::string const esimerkkijono = "Katenointi "
                    "on ominaisuus jolla kääntäjä"
                    " yhdistää peräkkäin olevat"
                    " merkkijonoliteraalit yhdeksi.\n";

std::vector<unsigned long int>
laskeTaulukkoVastaus( unsigned int pituus,
                      std::map<unsigned long int,
                      std::string> const& tiedot );
```

5.4. Osoittimen ja viitteen merkki kuuluvat tyyppinimen yhteyteen.

```
Paivays* ptr = NULL;
void tulostaPaivays( Paivays& paivaysOlio );
```

5.5. Primitiivejä **if**, **else**, **while**, **for** ja **do** seuraa aina koodilohko, joka on sisennetty johdonmukaisesti välilyöntimerkeillä.

> *Sarkaimen (tabulator) käyttö ei ole suositeltavaa, koska erilaiset tekstieditorit ja katseluohjelmat näyttävät sen sisennyksen eri tavoin ja voivat tuottaa lukukelvottoman lopputuloksen koodista, joka alkuperäisessä kirjoitusympäristössä on näyttänyt selkeältä.*

5.6. Koodilohkon alku- ({) ja loppumerkki (}) sijoitetaan omalle rivilleen samalle sarakkeelle (tai jollakin muulla yhtenäisellä ja selkeällä tavalla eroteltuna muusta koodista).

5.7. Jokaisessa **switch**-lauseessa on **default**-määre.

5.8. Jokainen **switch**-lauseen **case**-määreen toteutus on oma koodilohkonsa ja se päättyy lauseeseen **break** (samalla toteutuksella voi olla useita **case**-määreitä).

```
switch( merkki )
{
  case 'a':
  case 'A':
  {
      ⋮
    break;
  }

  default:
  {
    break;
  }
}
```

6. C-kielen rakenteet C++:ssa

6.1. Ohjelmasta ei saa poistua funktiolla `exit()` tai `abort()`.

*Nämä funktiot lopettavat koko ohjelman suorituksen ilman, että ohjelman kaikki oliot tuhoutuvat hallitusti (niiden purkajat jäävät suorittamatta). Oikea tapa on aina poistua funktiosta `main()` normaalisti (**`return`**-lauseella).*

6.2. `main()`-funktion paluuarvo on aina **`int`**.

ISO C++ -standardin määrittelemä ainoa mahdollinen paluuarvo.

6.3. `main()`-funktiota ei saa kutsua, siitä ei saa ottaa funktio-osoitinta, sitä (sen nimeä) ei saa kuormittaa eikä se saa olla **`inline`** eikä staattinen funktio — toisin sanoen funktionimeä `main` ei saa käyttää kuten tavallista funktiota.

ISO C++ -standardi käsittelee `main()`-funktion erikoistapauksena (se ei siis ole tavallinen funktio), ja kieltää nämä käyttötavat.

6.4. **`goto`**-lausetta ei saa käyttää.

Ohjelman normaalin toiminnan hallintaan on olemassa parempia kontrollirakenteita ja virhetilanteissa "hyppääminen muualle" toteutetaan poikkeusten avulla.

6.5. Käytä tietovirtoja (`cin`, `cout`, `cerr` ja `clog`) C-kielen IO-funktioiden sijaan.

Tietovirtojen operaatiot tekevät tyyppitarkistuksia, joita esimerkiksi C-funktio `printf()` ei tee.

6.6. Funktioita `malloc()`, `realloc()` ja `free()` ei saa käyttää, vaan niiden sijasta käytetään operaattoreita **`new`** ja **`delete`** [Rintala ja Jokinen, 2005, aliluku 3.2] [aliluku 3.5].

Uudet operaattorit kutsuvat olioiden dynaamisen käsittelyn yhteydessä rakentajia ja purkajia, joista vanhat muistinvarausrutiinit eivät tiedä mitään.

6.7. C:n kirjastoja käytettäessä niiden esittelyt tulee ottaa käyttöön määreellä **extern** *"C"*.

Vain tällä merkinnällä varmistetaan yhteensopivuus C-kielisen kirjaston kutsumekanismin kanssa.

```
extern "C" {
#include <gtk/gtk.h>
#include <curses.h>
}
```

7. Tyyppimuunnokset

7.1. Mikään osa ohjelmakoodista ei saa luottaa implisiittiseen tyyppimuunnokseen.

7.2. Ohjelmassa tulee käyttää kielen uusia tyyppimuunnosoperaattoreita (**static_cast**, **dynamic_cast** ja **reinterpret_cast**) vanhempien tyyppimuunnosten sijaan [Rintala ja Jokinen, 2005, aliluku 7.4.1].

Uudet tyyppimuunnosoperaattorit tekevät tarkistuksia muunnosten "järkevyydestä", jota vanhat muunnokset eivät tee.

7.3. Vakio-ominaisuutta ei saa poistaa tyyppimuunnoksella (Operaattoria **const_cast** ei saa käyttää) [Rintala ja Jokinen, 2005, aliluku 7.4.1].

Vakio on suunnitteluvaiheessa määrätty ominaisuus, jota ei pitäisi olla tarvetta poistaa ohjelmointivaiheessa.

8. Funktiot ja jäsenfunktiot

8.1. Funktion parametrien nimet on annettava sekä esittelyssä että määrittelyssä ja niiden on oltava molemmissa samat.

8.2. Funktion olioparametrin välitykseen käytetään (vakio)viitettä aina kun se on mahdollista [Rintala ja Jokinen, 2005, aliluku 4.3] [aliluku 7.3].

Näin vältetään olioiden turhaa kopiointia.

8.3. Funktiolla ei saa olla määrittelemätöntä parametrilistaa (ellipsis notation).

Määrittelemätön parametrilista poistaa käytöstä kaikki kääntäjän tekemät tarkistukset parametrinvälityksessä.

8.4. Funktion mahdolliset oletusparametrit ovat näkyvissä esittelyssä eikä niitä saa lisätä määrittelyssä.

8.5. Funktion paluuarvo on aina määriteltävä.

8.6. Funktio ei saa palauttaa osoitinta tai viitettä sen omaan paikalliseen dataan.

Paikallinen data ei ole kielen määrittelyn mukaan enää käytettävissä kun funktiosta on palattu.

8.7. Julkisen rajapinnan jäsenfunktio ei saa palauttaa olion jäsenmuuttujaan viitettä eikä osoitinta (vakioviite ja vakio-osoitin käy) [Rintala ja Jokinen, 2005, aliluku 4.2].

Olion tilaa on tarkoitus käsitellä ainoastaan rajapintafunktioiden avulla ja osoitin/viite antaisi mahdollisuuden sisäisen arvon muutokseen ilman rajapintakutsua.

8.8. Jäsenfunktiosta tehdään vakiojäsenfunktio aina kun se on mahdollista [Rintala ja Jokinen, 2005, aliluku 4.3].

9. Luokat ja oliot

9.1. Oliot toteutetaan C++:n rakenteella **`class`** ja tietueet (esimerkiksi linkitetyn listan yksi alkio) toteutetaan rakenteella **`struct`**.

Näin erotellaan selkeästi tietueet (rakenteinen tietorakenne) olioista.

9.2. Tietueella saa olla rakentajia ja virtuaalinen toteutukseltaan tyhjä purkaja, muttei muita jäsenfunktioita.

Tietueissä säilytetään yhteenkuuluvaa tietoa, jolloin niiden ainoa sallittu toiminnallisuus liittyy uuden tietueen alustukseen ja tietueen tuhoutumiseen (virtuaalipurkaja tarvitaan periytetyn tietueen oikean tuhoutumisen varmistamiseksi).

9.3. Luokan osat **`public`**, **`protected`** ja **`private`** esitellään tässä järjestyksessä [Rintala ja Jokinen, 2005, aliluku 4.2].

Luokan käyttäjää kiinnostaa ainoastaan julkisessa rajapinnassa olevat palvelut, joten niille selkein paikka on heti luokkaesittelyn alussa.

9.4. Luokan jäsenmuuttujat ovat aina näkyvyydeltään **`private`** [Rintala ja Jokinen, 2005, aliluku 4.2].

Modulaarisuuden kapselointiperiaatteen toteutus C++-luokassa.

9.5. Luokan esittelyssä ei koskaan ole toteutusta (toteutus mahdollisista **`inline`**-jäsenfunktioista on otsikkotiedostossa luokkaesittelyn jälkeen) [Rintala ja Jokinen, 2005, aliluku 2.3.3].

Esittelyssä on nimensä mukaisesti tarkoitus ainoastaan esitellä jokin rakenne. Sen toteutusyksityiskohdat ovat muualla eikä rakenteen käyttäjän tarvitse niitä nähdä. Katso myös kohdat 2.3.4 ja 2.3.5.

9.6. Luokalla ei pääsääntöisesti saa olla ystäväfunktioita eikä ystäväluokkia (**friend**-ominaisuus) [Rintala ja Jokinen, 2005, aliluku 8.4].

*"Ystävyys" rikkoo olioiden kapselointiperiaatetta vastaan. Yleinen poikkeus säännölle ovat kiinteästi yhteenkuuluvan luokkaryppään keskinäiset erioikeudet esimerkiksi saman ohjelmakomponentin sisällä. C++:ssa tulostusoperaattori **operator** << on pakko toteuttaa luokasta erillisenä funktiona, jolloin siitä usein tehdään ystäväfunktio (jos suunnittelussa tulostusta pidetään luokan julkisen rajapinnan operaationa).*

9.7. Luokalle kuormitettavien operaattoreiden semantiikka on säilytettävä (esimerkiksi **operator** + tekee aina yhteenlaskua "vastaavan" toiminnon).

10. Elinkaari

10.1. Muistinhallinta

10.1.1. Dynaamisesti varatun muistin vapauttaminen on pääsääntöisesti sen komponentti vastuulla (olio tai moduuli), joka varasi muistin [Rintala ja Jokinen, 2005, aliluku 3.5.4].

10.1.2. Operaattorilla **delete** saa vapauttaa ainoastaan muistin, joka on varattu operaattorilla **new** (**delete**n parametri saa olla vain **new**:n palauttama osoitin tai arvo nolla) [Rintala ja Jokinen, 2005, aliluku 3.5.2].

Tämän määrittelee ISO C++ -standardi.

10.1.3. Operaattorilla **delete**[] saa vapauttaa ainoastaan muistin, joka on varattu operaattorilla **new**[] [Rintala ja Jokinen, 2005, aliluku 3.5.3].

ISO C++ -standardi.

10.1.4. Jos **delete**-lauseen parametri on osoitinmuuttuja, johon voi sijoittaa, sen arvoksi sijoitetaan nolla **delete**ä seuraavassa lauseessa [Rintala ja Jokinen, 2005, aliluku 3.5.4].

Tuhotun olion käyttö myöhemmin saman osoittimen läpi tehdään näin mahdottomaksi (useimmat ajoympäristöt tuottavat virheen, jos nollaosoitinta yritetään käyttää).

```
Paivays* p = new Paivays;
   ⋮
delete p; p = 0;
```

10.2. Olion luonti ja tuhoaminen

10.2.1. Dynaamisesti (operaattorilla **new**) luodun olion tuhoutuminen varmistetaan `auto_ptr`-rakenteella (tai muilla älykkäillä osoitintoteutuksilla) aina kun sen käyttö on mahdollista [Rintala ja Jokinen, 2005, aliluku 3.5] [aliluku 11.7]. (Mutta esimerkiksi kohdan 12.3.4 sääntö estää `auto_ptr`:n taltioinnin STL:n säiliöihin.)

10.2.2. Jokaiselle luokalle on määriteltävä vähintään yksi rakentaja [Rintala ja Jokinen, 2005, aliluku 3.4.1].

Kääntäjän luoma toteutukseltaan tyhjä oletusrakentaja on äärimmäisen harvoin hyödyllinen alustus oliolle.

10.2.3. Rakentajissa ja purkajissa ei saa kutsua virtuaalifunktioita [Rintala ja Jokinen, 2005, aliluku 6.5.7].

Dynaaminen sitominen ei toimi totutulla tavalla kun virtuaalifunktiota kutsutaan rakentajan tai purkajan toteutuksesta.

10.2.4. Rakentajissa ja purkajissa ei saa käyttää globaaleja eikä staattisia olioita.

Olio voi itse olla staattinen tai globaali eivätkä sen käyttämät muut vastaavan tason oliot ole vielä välttämättä olemassa alustettuna kun rakentajan suoritus alkaa. Vastaavasti purkajassa voidaan vahingossa viitata jo aiemmin tuhottuun olioon.

10.2.5. Jos rakentajan toteutuksessa (toteutustiedostossa) tulostetaan, ennen luokan esittelyä (otsikkotiedosto) pitää ottaa käyttöön otsikkotiedosto <`iostream`>.

Ainoastaan tällä tavalla voidaan varmistua siitä, että tulostusolio (esim. `cout`) on olemassa ennen luokasta tehtyä oliota.

10.2.6. Yksiparametrinen rakentaja on merkittävä **`explicit`**-määreellä [Rintala ja Jokinen, 2005, aliluku 7.4.2], ellei se erityisesti ole tarkoitettu implisiittiseksi tyyppikonversioksi. ***Poikkeuksena kopiorakentaja, jolle tätä määrettä ei saa laittaa.***

Tämä estää kääntäjää yrittämästä käyttää rakentajaa implisiittisenä tyyppimuunnoksena. (Kohta 7.1.)

10.2.7. Itse tehty kantaluokan purkaja on aina **`public`** ja virtuaalinen tai **`protected`** ja ei-virtuaalinen. [Rintala ja Jokinen, 2005, aliluku 6.5.5].

*Periytetyn luokan olio tuhotaan oikein kantaluokkaosoitinta käyttävässä **`delete`**-operaatiossa ainoastaan, jos purkaja on virtuaalifunktio. Sijoittamalla purkaja **`protected`**-puolelle saadaan kantaluokkaosoittimen kautta tehtävä **`delete`** tehtyä mahdottomaksi.*

10.2.8. Purkajan tulee vapauttaa kaikki tuhoutumishetkellä olion vastuulla olevat resurssit [Rintala ja Jokinen, 2005, aliluku 3.4.2].

10.3. Olion kopiointi

10.3.1. Luokalle esitellään aina kopiorakentaja [Rintala ja Jokinen, 2005, aliluku 7.1.2].

Kääntäjän tuottama oletustoiminnallisuus ei välttämättä tuota halutunlaista kopiota oliosta. Katso myös kohta 10.3.2.

10.3.2. Tarpeettoman kopiorakentajan käyttö estetään esittelemällä se ilman toteutusta luokan **`private`**-rajapintaan [Rintala ja Jokinen, 2005, aliluku 7.1.2].

Luokan ulkopuolinen käyttöyritys tuottaa näin käännösvirheen ja luokan sisäinen kutsu paljastuu ohjelmiston linkitysvaiheessa (toteutus puuttuu).

10.3.3. Periytetyn luokan kopiorakentajan tulee kutsua kantaluokan kopiorakentajaa alustuslistassaan [Rintala ja Jokinen, 2005, aliluku 7.1.2].

Kopioinnin semantiikassa haluttu toiminnallisuus, jota kääntäjä ei tee automaattisesti.

10.4. Olion sijoitus

10.4.1. Jokaisessa luokassa esitellään oma sijoitusoperaattori [Rintala ja Jokinen, 2005, aliluku 7.2.2].

Kääntäjän tuottama oletustoiminnallisuus ei välttämättä tuota halutunlaista sijoitusta olioiden välillä.

10.4.2. Tarpeettoman sijoitusoperaattorin käyttö estetään esittelemällä se ilman toteutusta luokan **`private`**-osassa [Rintala ja Jokinen, 2005, aliluku 7.2.2].

Vertaa kohta 10.3.2.

10.4.3. Periytetyn luokan sijoitusoperaattorin tulee kutsua kantaluokan sijoitusoperaattoria [Rintala ja Jokinen, 2005, aliluku 7.2.2].

Sijoituksen semantiikassa haluttu toiminnallisuus, jota kääntäjä ei tee automaattisesti.

10.4.4. Sijoitusoperaattorin tulee aina varautua “itseen sijoitamista” (`x = x`) vastaan [Rintala ja Jokinen, 2005, aliluku 7.2.2].

“Turha operaatio”, joka saattaa sekoittaa muutoin oikein toimivan sijoitusoperaattorin.

```
Paivays& Paivays::operator =( Paivays const& rhs )
{
  if( this != &rhs )
  { // EI sijoiteta olioon itseensä
      ⋮
  }
  return *this;
}
```

10.4.5. Sijoitusoperaattorin paluuarvo on ***this** [Rintala ja Jokinen, 2005, aliluku 7.2.2].

Tämä ominaisuus ylläpitää sijoitusoperaattorin yleistä semantiikkaa, joka mahdollistaa esimerkiksi ketjusijoituksen “`a=b=c;`”.

11. Periytyminen

11.1. Ainoastaan **public**-periytymisen käyttö on sallittua mallintamaan olio-ohjelmoinnin periytymistä [Rintala ja Jokinen, 2005, aliluku 6.3].

*Oliosuunnittelumenetelmien käyttämä käsite “periytyminen” tarkoittaa C++-kielen **public**-periytymistä. Muut C++:n vaihtoehdot eivät tätä ole.*

11.2. Periytyminen on staattinen “is-a”-suhde. “has-a” toteutetaan jäsenmuuttujilla ja tyyppiriippumaton koodi malleilla [Rintala ja Jokinen, 2005, aliluku 6.3.4].

11.3. Moniperiytymistä saa käyttää ainoastaan rajapinnan tai toiminnallisen yksikön periyttämiseen (interface ja mixing) [Rintala ja Jokinen, 2005, aliluku 6.3].

Moniperiytyminen on monimutkainen ominaisuus, jota harvoin käytetään oliosuunnittelussa. Rajapintojen periyttäminen taas on paljon käytetty osa oliosuunnittelua.

11.4. Rajapintaluokan kaikki jäsenfunktiot ovat puhtaita virtuaalifunktioita (pure virtual) [Rintala ja Jokinen, 2005, aliluku 6.6].

Rajapinnan tarkoitus on esitellä operaatiot ja C++:n puhdas virtuaalifunktio on ominaisuus, joka pakottaa näiden toteutuksen periytetyssä luokassa.

11.5. Periytetyn luokan rakentajan alustuslistassa täytyy kutsua kantaluokan rakentajaa [Rintala ja Jokinen, 2005, aliluku 6.3.2].

Varmistetaan, että kantaluokka alustetaan halutulla tavalla (kääntäjän kutsuma oletusrakentaja kantaluokalle ei välttämättä ole oikea tapa).

11.6. Kantaluokan määrittelemiin virtuaalifunktioihin, jotka peritetty luokka määrittelee uudelleen, liitetään avainsana **`virtual`** myös periytetyssä luokassa [Rintala ja Jokinen, 2005, aliluku 6.5.5].

11.7. Periytynyttä ei-virtuaalista jäsenfunktiota ei saa määritellä uudelleen [Rintala ja Jokinen, 2005, aliluku 6.5].

Vaikka C++ sallii tämän operaation, sillä ei ole oliosuunnittelun kannalta mitään perusteltua käyttöä.

11.8. Periytynyttä virtuaalisen jäsenfunktion oletusparametria ei saa määritellä uudelleen.

12. Mallit

12.1. Mallin tyyppiparametreilleen asettamat vaatimukset tulee dokumentoida mallin esittelyn yhteydessä [Rintala ja Jokinen, 2005, aliluku 9.5.4].

Tyyppiparametrin vaatimukset ovat tarkistuslista mallin käyttäjälle, että mallia on käytetty oikealla tavalla. Nämä vaatimukset eivät useinkaan näy selkeästi mallin rakenteessa ja kääntäjän niistä antamat virheilmoitukset voivat olla parhaimmillaankin sekavia.

12.2. Mallin käyttäjille tulisi tarvittaessa tarjota valmiiksi optimoituja toteutuksia mallin erikoistusten avulla [Rintala ja Jokinen, 2005, aliluku 9.5.6].

Mallit ovat yleensä uudelleenkäytettäviksi tarkoitettuja kirjastorutiineja, joiden käytettävyyttä helpottavat niiden monipuoliset ja tehokkaat toteutukset.

12.3. STL

12.3.1. `C`-kielen vanhan taulukkotyypin sijaan tulee käyttää STL:n sarjasäiliöitä `vector` tai `deque` [Rintala ja Jokinen, 2005, aliluku 10.2] [aliluku A.3].

Uudet rakenteet ovat helppokäyttöisempiä. Jos jokin (vanhan ohjelmakoodin) funktio tarvitsee esimerkiksi parametrikseen taulukon, sille voidaan antaa osoitin vektorin alkuun (`&v[0]`).

12.3.2. STL:n sisäisestä toiminnasta ei saa tehdä oletuksia. Ainoastaan ISO `C++`:n määrittelemiä ominaisuuksia saa hyödyntää.

12.3.3. Käytetyille STL-säiliöille annetaan kuvaavat nimet tyyppialiaksen avulla.

```
typedef std::map<std::string,unsigned int>
        Puhelinluettelo;
```

12.3.4. STL:n säiliöön saa tallettaa ainoastaan olioita, joilla on kopiorakentaja ja sijoitusoperaattori, jotka tuottavat uuden tilaltaan identtisen olion [Rintala ja Jokinen, 2005, aliluku 10.2].

Säiliöt voivat sisäisesti siirrellä olioita kopioimalla tai sijoittamalla ja nämä operaatiot eivät saa sotkea taltioituja olioita.

12.3.5. Käytettäessä STL:n säiliöitä on varmistuttava siitä, että käytettyjen operaatioiden pahimman tapauksen ajankäyttö on riittävän tehokasta ohjelmiston toiminnan kannalta [Rintala ja Jokinen, 2005, aliluku 10.1].

Ajankäyttöominaisuudet ovat tärkein peruste tehtäessä valintaa esimerkiksi eri sarjasäiliöiden välillä. Paras vaihtoehto riippuu usein siitä, mitä operaatioita säiliölle tehdään ohjelmassa eniten.

12.3.6. STL:n säiliöiden yhteydessä tulee käyttää STL:n algoritmeja omien toteutusten sijaan [Rintala ja Jokinen, 2005, aliluku 10.4].

STL:n algoritmit ovat luultavasti omia toteutuksia tehokkaampia ja ne pystyvät tekemään sisäisiä optimointeja.

12.3.7. STL:n säiliön alkioden läpikäynti toteutetaan iteraattoreiden avulla (`begin()`–`end()` -väli) [Rintala ja Jokinen, 2005, aliluku 10.3].

Esimerkki funktiosta, joka käy läpi kaikki säiliön alkiot riippumatta siitä mitä STL:n säiliöistä on käytetty parametrin toteutukseen:

```
void tulostaPuhelinluettelo(Puhelinluettelo& tk)
{
  Puhelinluettelo::const_iterator alkio;
  for (alkio = tk.begin(); alkio != tk.end(); ++alkio)
  // Vertailu "alkio < tk.end()" ei toimi
  // kaikkilla säiliöillä (niiden iteraattoreilla)
  {
    std::cout << *alkio << std::endl;
  }
}
```

12.3.8. Iteraattorin lisääminen ja vähentäminen suoritetaan etuliiteoperaattorilla (preincrement ja predecrement) [Rintala ja Jokinen, 2005, aliluku 10.3].

Lausekkeen evaluoinnin jälkeen suoritettava operaatio (postincrement) on tehottomampi, koska se joutuu palauttamaan kopion iteraattorin vanhasta arvosta.

12.3.9. Mitätöitynyttä iteraattoria ei saa käyttää (ainoastaan tuhoaminen ja uuden arvon sijoittaminen ovat sallittuja operaatioita) [Rintala ja Jokinen, 2005, aliluku 10.3].

ISO C++ -standardi.

13. Poikkeukset

13.1. Poikkeuksia saa käyttää ainoastaan virhetilanteiden ilmaisemiseen [Rintala ja Jokinen, 2005, aliluku 11.4].

Poikkeusten käsittely on muita kontrollirakenteita raskaampaa, joten niitä ei kannata käyttää ohjelmiston normaalin kontrollin ohjaamiseen.

13.2. Jos poikkeuskäsittelijä ei pysty täysin toipumaan sieppaamastaan poikkeuksesta, se "heitetään" uudelleen ylemmälle tasolle [Rintala ja Jokinen, 2005, aliluku 11.4.2].

Esimerkiksi funktion paikallinen poikkeuskäsittelijä vapauttaa kaikki ennen poikkeuksen ilmoittamaa virhettä varatut resurssit ja ilmoittaa virheestä ylemmälle tasolle heittämällä saman poikkeuksen eteenpäin (`throw`-lause ilman parametria).

13.3. Pääohjelmasta ei saa vuotaa poikkeuksia ulos [Rintala ja Jokinen, 2005, aliluku 11.4.2].

Useat ajoympäristöt antavat erittäin epäselviä ilmoituksia ohjelmasta ulos vuotavien poikkeusten yhteydessä.

13.4. Luokan purkajasta ei saa vuotaa poikkeuksia [Rintala ja Jokinen, 2005, aliluku 11.5].

Poikkeusten käsittelyssä usein tuhotaan olioita, joiden purkajien mahdollisesti heittämä poikkeus aiheuttaisi kahden poikkeuksen voimassaolon yhtäaikaa. Tässä tilanteessa ohjelman suoritus lopetetaan ilman jatkokäsittelyä, joka ei yleensä ole haluttu toiminnallisuus.

13.5. Pääsääntöisesti funktioiden poikkeusmäärelistaa ei saa käyttää. Vain jos funktion kaikissa käyttötilanteissa varmasti tiedetään siitä mahdollisesti vuotavat poikkeukset, voi poikkeukset luetella funktioesittelyn poikkeuslistassa. [Rintala ja Jokinen, 2005, aliluku 11.6].

Varsinkin virtuaalifunktioiden ja periytymisen yhteydessä jäsenfunktio ei pysty luettelemaan kaikkia poikkeuksia mitä kyseisen funktion kutsun yhteydessä mahdollisesti tulee (nämä poikkeukset

saattaa määritellä periytynyt luokka). Usein on myös suunnittelun kannalta järkevää antaa poikkeusten vuotaa "välitason" rajapinnasta ylemmälle ohjelmiston tasolle.

Viitteet

[Rintala ja Jokinen, 2005] Matti Rintala ja Jyke Jokinen. *Olioiden ohjelmointi C++:lla, 4., uudistettu painos*. Talentum, 2005. ISBN 952-14-0936-3.