C-kielestä C++-kieleen

C++-kieleen tottuminen kestää jonkin aikaa itse kultakin, mutta harmaantuneille C-ohjelmoijille prosessi voi olla erityisen hermoja raastava. Koska C-kieli on tehokkaasti C++-kielen osajoukko, kaikki vanhat C-temput toimivat edelleen, mutta ei enää ole sopivaa käyttää monia niistä. Esimerkiksi osoitin osoittimeen näyttää C++-ohjelmoijien mielestä hieman hassulta. Miksi ei ole käytetty *viittausta* osoittimeen?

C on suhteellisen yksinkertainen kieli. Se sisältää itse asiassa vain makroja, osoittimia, struktuureja, taulukoita ja funktioita. Oli ongelma mikä tahansa, niin ratkaisu johtaa aina makroihin, osoittimiin, struktuureihin, taulukoihin ja funktioihin. C++-kielessä asia on toisin. Makrot, osoittimet, rakenteet ja taulukot ovat edelleen tallella, mutta niin ovat myös yksityiset ja suojatut jäsenet, funktioiden kuormittaminen, oletusparametrit, muodostinfunktiot ja tuhoajafunktiot, käyttäjän määrittelemät operaattorit, avoimet funktiot, viittaukset, ystävät, mallit, poikkeukset, nimiavaruudet ja paljon muuta. C++-kielen suunnittelutila on paljon rikkaampi kuin C-kielessä: tarjolla on yksinkertaisesti paljon enemmän harkittavissa olevia vaihtoehtoja.

Kun vastaan tulee tällainen määrä vaihtoehtoja, monet C++-ohjelmoijat lyyhistyvät ja pysyttelevät tiukasti kiinni siinä, mihin ovat tottuneet. Tämä ei useimmiten ole mikään suuri synti, mutta eräät C-tottumukset toimivat C++-kielen hengen vastaisesti. Näiden tottumusten vain *täytyy* poistua.

Kohta 1: Suosi `const-` ja `inline`-avainsanoja `#define`-esikääntäjäkomennon sijasta.

Tätä Kohtaa voitaisiin paremmin kutsua otsikolla "suosi kääntäjää esikääntäjän sijasta", koska `#define`-esikääntäjäkomentoa kohdellaan usein kuin se *itse* ei olisi ollenkaan osa kieltä. Tämä on yksi ongelmista. Kun kirjoitat jotain tämänkaltaista

```
#define ASPECT_RATIO 1.653
```

voi olla, että kääntäjät eivät koskaan tule näkemään symbolista nimeä `ASPECT_RATIO`. Esikääntäjä voi poistaa sen ennen kuin lähdeteksti koskaan saa-

vuttaa kääntäjän. Tästä on tuloksena se, että ASPECT_RATIO-nimeä ei ehkä koskaan kirjoiteta symboliseen taulukkoon. Tämä voi olla hämmentävää, varsinkin, jos saat käännöksen aikana tämän vakion käyttöä koskevan virheilmoituksen, koska se voi viitata lukuun 1.653, ei ASPECT_RATIO-nimeen. Jos ASPECT_RATIO on määritetty otsikkotiedostossa, jota et kirjoittanut, sinulla ei silloin ole aavistustakaan mistä tuo 1.653 alun perin tuli, ja aikaa kuluu luultavasti hukkaan sen jäljittämisessä. Tämä ongelma voi myös tulla esiin symbolisessa virheenkäsittelijässä, koska ohjelmoitu nimi ei ehkä taaskaan ole symbolisessa taulukossa.
Ratkaisu tähän valitettavaan skenaarioon on yksinkertainen ja lakoninen. Sen sijaan, että käyttäisit esikääntäjän makroa, määritä vakio:

```
const double ASPECT_RATIO = 1.653;
```

Tämä näkemys toimii ihanteellisesti. On kuitenkin kaksi mainitsemisen arvoista erikoistapausta.

Ensinnäkin asiat voivat mutkistua, kun määritetään vakio-osoittimia. Koska vakioiden määritykset on tavallisesti sijoitettu otsikkotiedostoihin (ne sisältyvät useisiin lähdeteksteihin), on tärkeää, että *osoitin* esitellään `const`-tyyppisenä, lisättynä yleensä siihen, mihin osoitin osoittaa. Silloin, kun esimerkiksi määritetään vakio, joka on `char*`-pohjainen merkkijono otsikkotiedostossa, sinun pitää kirjoittaa `const`-avainsana kahdesti:

```
const char * const authorName = "Scott Meyers";
```

Jos haluat lisää tietoa `const`-määreen tarkoituksesta ja käytöstä varsinkin osoittimien yhteydessä, katso Kohta 21.

Toiseksi luokkakohtaisten vakioiden määrittäminen on usein asianmukaista, ja tämä ohjaa hieman eri suuntaan. Jos vakion näkyvyysalue halutaan rajoittaa luokkaan, siitä pitää tehdä jäsen, ja jos haluat varmistaa, että vakiosta on ainakin yksi kopio, sinun täytyy tehdä siitä *static*-tyyppinen jäsen:

```
class GamePlayer {
private:
  static const int NUM_TURNS = 5;    // vakion esittely
  int scores[NUM_TURNS];             // vakion käyttö
  ...
};
```

On kuitenkin pieni ryppy, ja se on, että se minkä näet alla, on NUM_TURNS-vakion *esittely*, ei määritys. Luokan staattiset jäsenet täytyy silti määrittää toteutustiedostossa:

```
const int GamePlayer::NUM_TURNS;  // pakollinen määritys;
                                  // sij. luokan toteut.tied.
```

Sinun ei kannata menettää yöuniasi murehtiessasi tätä yksityiskohtaa. Jos unohdat määrityksen, linkkerisi pitäisi muistuttaa sinua.

Vanhemmat kääntäjät eivät ehkä hyväksy tätä kielioppia, koska alustavan arvon määritys sen esittelyvaiheessa oli aikoinaan laitonta luokan staattiselle jäsenelle. Luokan sisäinen alustus sallitaan lisäksi vain sisäisille tyypeille (eli `int`, `bool`, `char` jne), ja vain vakioille. Niissä tapauksissa, joissa yllä olevaa kielioppia ei voida käyttää, alustusarvo sijoitetaan määrityksen vaiheessa:

```
class EngineeringConstants {   // tämä sijoitetaan luokan
private:                       // otsikkotiedostoon

  static const double FUDGE_FACTOR;

  ...

};

// tämä sijoitetaan luokan toteutustiedostoon
const double EngineeringConstants::FUDGE_FACTOR = 1.35;
```

Tämä riittää useimmissa tapauksissa. Ainoa poikkeus on silloin, kun tarvitset luokkavakion arvon luokan kääntämisen aikana, kuten silloin, kun esittelet yllä olevan taulukon `GamePlayer::scores`-funktion (tällöin kääntäjien pitää tietää taulukon koko käännöksen aikana). Tällöin hyväksytty tapa hyvittää kääntäjille, jotka (väärin) kielsivät luokan sisäisen alustavien arvojen määrityksen kokonaisuuteen kuuluville luokkavakioille, on käyttää sitä, mikä lempeästi tunnetaan termillä "the enum hack". Tämä tekniikka hyötyy siitä tosiseikasta, että luetellun tyypin arvoja voidaan käyttää siellä, missä odotetaan `int`-tyyppisiä arvoja, joten `GamePlayer` olisi aivan hyvin voitu määrittää tällä tavalla:

```
class GamePlayer {
private:
  enum { NUM_TURNS = 5 };    // "the enum hack"— tekee
                             // NUM_TURNS symbolisen nimen
                             // arvolle 5

  int scores[NUM_TURNS];     // toimii

  ...

};
```

Sinun ei pitäisi tarvita käyttää "enum hack" -käsitettä, paitsi jos työskentelet pääasiassa historiallisesti kiinnostavilla kääntäjillä (eli niillä, jotka on kirjoitettu ennen vuotta 1995). Kannattaa silti tietää miltä se näyttää, koska on aika yleistä törmätä siihen lähdetekstissä, joka on päivätty näihin aikaisempiin, yksinkertaisempiin aikoihin.

Jos palaamme esikääntäjään, toinen yleinen `#define`-esikääntäjäkomennon (väärin)käyttö tapahtuu silloin, kun toteutetaan makroja, jotka näyttävät funktioilta, mutta niille ei koidu kustannuksia funktiokutsun kutsumisesta. Kanoninen esimerkki tästä on kahden luvun enimmäisarvojen laskeminen:

```
#define max(a,b) ((a) > (b) ? (a) : (b))
```

Tämä esimerkki sisältää niin paljon epäkohtia, että pelkkä ajatteleminen tuottaa tuskaa. Olisi parempi leikkiä moottoritiellä ruuhka-aikana.

Aina kun kirjoitat tämänkaltaisen makron, sinun pitää muistaa, että kun kirjoitat makron runkoa, sijoita kaikki argumentit sulkeisiin; muuten voit joutua vaikeuksiin, kun makroa kutsutaan lauseella. Mutta vaikka se menisikin oikein, katso mitä outoja asioita voi tapahtua:

```
int a = 5, b = 0;

max(++a, b);                    // a:n arvoa kasvatetaan //
kaksi kertaa
max(++a, b+10);                 // a:n arvoa kasvatetaan
                                // kerran
```

Se, mitä tässä tapahtuu `max`-funktion sisällä olevalle `a`-muuttujan arvolle, riippuu siitä, mihin sitä verrataan!

Onneksi tämän kaltaista järjettömyyttä ei tarvitse sietää. Voit saada makron kaiken tehokkuuden sekä odotettavan käyttäytymisen ja säännöllisen funktion tyyppiturvallisuuden käyttämällä avoimena määritettyä funktioa (katso Kohta 33):

```
inline int max(int a, int b) { return a > b ? a : b; }
```

Yllä oleva ei ole ihan sama kuin edellä oleva makro, koska `max`-funktion tätä versiota voidaan kutsua vain `int`-tyypeillä, mutta malli korjaa tuon ongelman aika mukavasti:

```
template<class T>
inline const T& max(const T& a, const T& b)
{ return a > b ? a : b; }
```

Tämä malli luo kokonaisen funktioperheen, joista jokainen vastaanottaa kaksi oliota, jotka ovat muunnettavissa samaan tyyppiin ja palauttavat viittauksen (eli vakioversion) suurempaan kahdesta oliosta. Koska et voi tietää, mitä tyyppiä `T`-luokka tulee olemaan, välität ja palautat tehokkuuden takia viittauksen (katso Kohta 22).

Muuten, ennen kuin harkitset mallien kirjoittamista yleisesti ottaen käytännöllisille funktioille, kuten `max`, tarkista ovatko ne jo standardikirjastossa (katso Kohta 49). Tulet esimerkiksi `max`-funktion kohdalla iloisesti yllättymään siitä, että voit levätä laakereillasi: `max` on osa standardia C++-kirjastoa.

Kun otetaan huomioon `const`-avainsanojen ja avointen funktioiden (`inline`-avainsana) saatavuus, esikääntäjän tarpeellisuus vähenee, mutta sen tarve ei poistu kokonaan. Se päivä on vielä kaukana, jolloin voit hylätä kokonaan `#include`- ja `#ifdef/#ifndef`-komennot, ja ne näyttelevät tärkeää roolia jatkossakin ohjatessaan kääntämistä. Esikääntäjä ei tule vielä jäämään eläkkeelle, mutta sinun tulee ehdottomasti antaa sille pitempiä ja säännöllisempiä lomia.

Kohta 2: Suosi `<iostream>`-otsikkotiedostoa `<stdio.h>`-otsikkotiedoston sijasta.

Kyllä, ne ovat siirrettävissä. Ne ovat myös tehokkaita. Tiedät myös jo, kuinka niitä käytetään. Kyllä kyllä kyllä. Niin palvottuja kuin ne ovatkin, tosiasia on, että `scanf`-funktio, `printf`-funktio ja muut vastaavat funktiot kaipaavat hieman parannusta. Erityisseikkana mainittakoon, että ne eivät ole tyyppiturvallisia eivätkä laajennettavissa. Koska tyyppiturvallisuus ja laajennettavuus on C++-elämäntavan kulmakiviä, niiden käytöstä voitaisiin aivan hyvin luopua. Sitä paitsi, `printf/scanf`-perheen funktiot erottavat ne muuttujat, jotka tullaan lukemaan tai kirjoittamaan muotoiluinformaatiosta, joka ohjaa lukemista ja kirjoittamista, aivan kuten FORTRAN-kielessä. On aika jättää haikeat jäähyväiset 1950-luvulle.

Ei ole yllätys, että `printf/scanf` -funktioiden heikkoudet ovat `operator>>`- ja `operator<<`-funktioiden vahvoja puolia.

```
int i;
Rational r;                    // r on suhdeluku

...

cin >> i >> r;
cout << i << r;
```

Kun tämä koodi käännetään, täytyy käyttää `operator>>`- ja `operator<<`-funktioita, jotka toimivat sen olion kanssa, joka on `Rational`-tyyppinen. Jos nämä funktiot puuttuvat, seurauksena on virhe. (`Int`-tyyppisten muuttujien versiot ovat standardi.) Kääntäjät huolehtivat lisäksi myös siitä, mitä versioita operaattoreista käytetään kutsuttaessa eri muuttujia, joten sinun ei tarvitse murehtia, onko ensimmäinen luettava ja kirjoitettava olio `int`-tyyppinen ja toinen `Rational`-tyyppinen.

Luettavat oliot käydään lisäksi läpi käyttämällä samaa lauseopillista muotoa kuin ne, jotka kirjoitetaan. Sinun ei siis tarvitse muistaa typeriä sääntöjä, kuten silloin, kun käytät `scanf`-funktioita. Tällöinhän sinun täytyy ottaa huomioon osoite, jos sinulla ei vielä ole osoitinta, mutta jos sinulla jo on osoitin, sinun täytyy varmistaa, että *et* määritä osoitetta. Anna C++-kääntäjien huolehtia näistä yksityiskohdista. Niillä ei ole parempaa tekemistä ja sinulla *on* muutakin tekemistä. Huomaa lopuksi, että sisäänrakennetut tyypit, kuten `int`, luetaan ja kirjoitetaan samalla tavalla kuin käyttäjän määrittelemät tyypit, kuten `Rational`. Yritäpä *tuota* käyttämällä `scanf`-funktiota ja `printf`-funktiota!

Tässä on tapa, jolla voisit kirjoittaa tulostusrutiinin luokalle, joka edustaa suhdelukuja:

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

  ...
```

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

friend ostream& operator<<(ostream& s, const Rational& r);
};

ostream& operator<<(ostream& s, const Rational& r)
{
  s << r.n << '/' << r.d;
  return s;
}
```

Tämä versio `Operator<<`-funktiosta esittää eräitä hienosyisiä (mutta tärkeitä) kohtia, joita käsitellään muualla tässä kirjassa. Esimerkiksi, `operator<<`-funktio ei ole jäsenfunktio (Kohdassa 19 selvitetään miksi), ja tulostettava `Rational`-olio välitetään `operator<<`-funktiolle mieluummin viittauksena `const`-tyyppiseen muuttujaan, kuin oliona (katso Kohta 22). Vastaava syöttöfunktio `operator>>` esiteltäisiin ja toteutettaisiin samalla tavalla.

Vaikka olen vastahakoinen myöntämään asian, on joitakin tilanteita, jolloin olisi järkevämpää luottaa kokeiltuun ja hyväksi havaittuun. Ensinnäkin eräät iostream-toimintojen toteutukset ovat vähemmän tehokkaita kuin vastaavat C-kielen virtaustoiminnot, joten on mahdollista (joskin epätodennäköistä), että sinulla on sovellus, johon tämä vaikuttaa ratkaisevasti. Kannattaa kuitenkin muistaa, että tämä ei *yleisesti* kerro mitään iostream-syöttövirtatoiminnoista, vain tietyissä toteutuksissa. Toiseksi, iostream-kirjasto muutettiin eräillä perustavanlaatuisilla tavoilla silloin, kun se oli matkalla kohti standardisointia (katso Kohta 49). Niinpä sovelluksissa, joiden täytyy maksimaalisesti olla siirrettävissä järjestelmästä toiseen, voidaan huomata, että eri jälleenmyyjät tukevat erilaisia approksimaatioita standardiin. Lopuksi, koska iostream-kirjaston luokilla on muodostinfunktioita ja `<stdio.h>`-otsikkotiedoston funktioilla ei ole, on harvinaisia tilanteita, jotka liittyvät staattisten olioiden alustusjärjestykseen (katso Kohta 47). Silloin perus-C-kirjasto voi olla paljon käytännöllisempi yksinkertaisesti siksi, että tiedät voivasi kutsua sitä aina ilman rangaistusta.

Iostream-syöttövirtakirjaston funktioiden ja luokkien tarjoama tyyppiturvallisuus ja laajennettavuus ovat paljon käytännöllisempiä asioita, kuin voisit alustavasti kuvitella, joten älä heitä niitä menemään siksi, että olet tottunut `<stdio.h>`-otsikkotiedostoon. Muistot ovat kuitenkin siirtymän jälkeen jäljellä.

Kohdan otsikossa ei muuten ole kirjoitusvirhettä: tarkoitan todella `<iostream>`, en `<iostream.h>`. Sellaista kuin `<iostream.h>` ei teknisesti puhuen ole olemassa - standardointikomitea eliminoi sen suosien `<iostream>`-otsikkoa silloin, kun se lyhensi niiden otsikkojen nimiä, jotka eivät ole standardia C-kieltä. Tähän selitetään syy Kohdassa 49, mutta sinun kannattaa ymmärtää vain se, että jos kääntäjäsi tukee sekä `<iostream>` että `<iostream.h>` -otsikkoja (kuten todennäköistä on), otsikot ovat aavistuksen verran erilaisia. Varsinkin, jos kirjoitat komennon `#include <iostream>`, saat iostream-tyyppisen kirjaston elementit piilotettuna `std`-nimiavaruuteen (katso Kohta 28), mutta jos kirjoitat komennon

`#include <iostream.h>`, saavutat nämä samat elementit globaalissa näkyvyysalueessa. Elementtien saaminen globaalissa näkyvyysalueessa voi johtaa nimeä koskeviin ristiriitoihin, tarkasti ottaen sen kaltaisiin ristiriitoihin, joiden estämiseksi nimiavaruuksien käyttö suunniteltiin. `<iostream>`-sanassa on sitä paitsi vähemmän kirjoitettavaa kuin `<iostream.h>`-sanassa. Monille ihmisille se riittää sen suosimiseen.

Kohta 3: Suosi `new`- ja `delete`-funktioita `malloc`- ja `free`-funktioiden sijasta.

Ongelma, joka koskee `malloc`- ja `free`-funktioita (ja niiden muunnelmia), on yksinkertainen: ne eivät tunne muodostinfunktioita ja tuhoajafunktioita.

Tutki kahta seuraavaa tapaa, joilla saat tilaa kymmenestä `string`-tyyppisestä oliosta koostuvalle taulukolle. Yksi käyttää `malloc`-funktiota ja muut `new`-funktioita:

```
string *stringArray1 =
  static_cast<string*>(malloc(10 * sizeof(string)));
string *stringArray2 = new string[10];
```

`stringArray1` osoittaa tässä muistiin, joka riittää kymmenelle `string`-oliolle, mutta johon ei ole muodostettu yhtään oliota. Taulukon olioiden alustamiseen ei tämän lisäksi ole mitään muuta keinoa, kuin käyttää eräitä melko hämäriä kielellisiä kiemuroita. `stringArray1` on toisin sanoen aika hyödytön. Vastakohtana sille, `stringArray2` osoittaa taulukkoon, jossa on kymmenen täysin muodostettua `string`-olioita, joista jokaista voidaan käyttää turvallisesti missä tahansa toiminnossa, joka vastaanottaa `string`-tyypin.

Olettakaamme, että olet joka tapauksessa taianomaisesti onnistunut alustamaan `stringArray1`-taulukon oliot. Myöhemmin ohjelmassa sinun odotetaan sitten tekevän jotain tämän kaltaista:

```
free(stringArray1);            // katso Kohdasta 5 miksi
delete [] stringArray2;        // "[]" ovat välttämättömät
```

`Free`-funktion kutsu vapauttaa muistin, johon `stringArray1` osoittaa, mutta muistissa oleviin `string`-olioihin ei kutsuta tuhoajafunktioita. Jos `string`-oliot ovat itse varanneet muistia, kuten niillä on taipumus tehdä, kaikki niiden varaama muisti menetetään. Toisaalta, kun `delete`-operaattoria kutsutaan `stringArray2`-taulukkoon, tuhoajafunktiota kutsutaan taulukon jokaiselle oliolle ennen kuin mitään muistia vapautetaan.

Koska `new`- ja `delete`-funktiot vaikuttavat toisiinsa sopivasti muodostinfunktioilla ja tuhoajafunktioilla, ne ovat selvästi suositeltavampi valinta.

`New-` ja `delete`-funktioiden sekoittaminen `malloc`- ja `free`-funktioiden kanssa on yleensä huono ajatus. Kun yrität kutsua `free`-funktiota osoittimeen, jonka vastaanotit `new`-funktiolla tai kutsut `delete`-funktiota osoittimeen, jonka vastaanotit `malloc`-funktiolla, tulokset ovat määrittelemättömät, ja tiedämme kaikki mitä "määrittelemätön" tarkoittaa: se tarkoittaa sitä, että ohjelma toimii kehitysvaiheessa, se toimii testausvaiheessa ja koko homma räjähtää kun olet kasvotusten tärkeimpien asiakkaitesi kanssa.

`new/delete`-funktioiden ja `malloc/free`-funktioiden yhteensopimattomuus voi johtaa eräisiin aika kiinnostaviin jälkivaikutuksiin. Esimerkiksi `strdup`-funktio, joka yleensä löytyy `<string.h>`-otsikosta, vastaanottaa `char*`-pohjaisen merkkijonon ja palauttaa kopion siitä:

```
char * strdup(const char *ps);    // pal. kop. mih. ps os.
```

Sekä C-kieli että C++-kieli käyttävät joskus samaa versiota `strdup`-funktiosta, joten funktion sisältä varattu muisti on `malloc`-funktion seurausta. Tuloksena C++-ohjelmoijat, jotka tietämättään kutsuvat `strdup`-funktiota, voivat jättää huomioimatta sen tosiasian, että heidän täytyy käyttää `free`-funktiota `strdup`-funktion palauttamassa osoittimessa. Mutta odota! `strdup`-funktio voidaan joskus päättää kirjoittaa uudelleen C++-kielellä estämällä jälkivaikutukset ja annetaan sitten tämän uudelleen kirjoitetun version kutsua funktion sisältä `new`-funktiota, valtuuttaen täten kutsujat käyttämään `delete`-funktiota myöhemmin. Kuten voit kuvitella, tämä voi johtaa eräisiin aika painajaismaisiin siirrettävyysongelmiin kun lähdetekstiä sukkuloidaan edestakaisin niiden paikkojen välillä, jotka käyttävät eri muotoa `strdup`-funktiosta.

C++-ohjelmoijat ovat silti yhtä kiinnostuneita lähdetekstin uudelleenkäyttämisestä kuin C-ohjelmoijat. On olemassa monia C-kirjastoja, jotka perustuvat `malloc`- ja `free`-funktioille ja sisältävät paljon lähdetekstiä, joka kannattaa ehdottomasti käyttää uudelleen. Kun hyödyt tällaisesta kirjastosta, on todennäköistä, että huomaat olevasi vastuussa siitä, että sinun täytyy vapauttaa kirjaston `malloc`-funktion varaama muisti `free`-funktiolla, ja/tai varata `malloc`-funktiolla muistia, jonka kirjasto itse vapauttaa `free`-funktiolla. Hieno homma. Ei ole ollenkaan väärin kutsua C++-ohjelman sisältä `malloc`- ja `free`-funktiota, kunhan varmistat, että `malloc`-funktiolla saadut osoittimet kohtaavat vapahtajansa `free`-funktiolla ja `new`-funktiolla saadut osoittimet vapautuvat lopulta `delete`-funktiolla. Ongelmat alkavat, kun rupeat hutiloimaan ja yrität sekoittaa `new`-funktiota `free`-funktion kanssa tai `malloc`-funktiota `delete`-funktion kanssa. Kun teet näin, *kaivat verta* nenästäsi.

Kun otetaan huomioon, että `malloc` ja `free` ovat tietämättömiä muodostinfunktioista ja tuhoajafunktioista, ja että `malloc/free`-funktioiden sekoittaminen `new/delete`-funktioiden kanssa voi olla yhtä haihtuvaa kuin ylioppilasyhdistyksen sisäänajobileet, on parempi, että pysyttelet aina kun pystyt muita funktioita hylkivällä dieetillä, joka koostuu `new-` ja `delete`-funktioista.

Kohta 4: Suosi C++-tyylisiä kommentteja.

C-kielen kommenteissa käytettävä vanha kunnon kielioppi toimii myös C++-kielessä, mutta C++-kielen uudelleenmuodostetulla kommentit-rivin-lopussa-kieliopilla on omat erityiset etunsa. Tutki esimerkiksi tätä tilannetta:

```
if ( a > b ) {
  // int temp = a;              // vaihda a ja b
  // a = b;
  // b = temp;
}
```

Tässä on lähdetekstilohko, joka on jostain kumman syystä kommentoitu, mutta huumaavana osoituksena ohjelmoinnin insinööritaidosta alkuperäisen lähdetekstin kirjoittanut ohjelmoija sisällytti siihen kommentin, joka osoitti, missä mennään. Silloin, kun C++-kielen kommentointimuotoa käytettiin lohkon kommentointiin, sisennettyä kommenttia ei otettu ollenkaan huomioon, mutta vakaviin ongelmiin olisi varmasti törmätty, jos joku olisi valinnut C-tyylisen kommentoinnin:

```
if ( a > b ) {
  /* int temp = a;              /* vaihda b ja a*/
     a = b;
     b = temp;
  */
}
```

Huomaa, kuinka sisennetty kommentti sijoittaa huomaamatta ennenaikaisen loppumerkin kommenttiin, jonka tarkoituksena oli kommentoida lähdetekstin lohko.

C-tyylisillä kommenteilla on silti oma tehtävänsä. Ne ovat esimerkiksi korvaamattomia sekä C-kielen että C++-kielen kääntäjien muodostamissa otsikkotiedostoissa. Jos kuitenkin käytät C++-tyylisiä kommentteja, sinun kannattaa pysytellä niissä.

Kannattaa korostaa, että taantuvat esikääntäjät, jotka oli kirjoitettu vain C-kielelle, eivät tiedä mitä tehdään C++-tyylisillä kommenteilla, joten alla oleva esimerkki ei aina toimi odotetulla tavalla:

```
#define LIGHT_SPEED 3e8         // m/sek (tyhjiössä)
```

Esikääntäjässä, jossa C++ ei ole tuttu, rivin lopussa olevasta kommentista tulee *osa makroa*! Kuten keskustelimme Kohdassa 1, esikääntäjää ei tietenkään pitäisi ollenkaan käyttää vakioiden määrittämiseen.