```
1 void PieniPaivays::sijoitaPaivays(PieniPaivays& p)
2 {
3   paiva_ = p.paiva_;
4   kuukausi_ = p.kuukausi_;
5   vuosi_ = p.vuosi_;
6 }
```

LISTAUS 4.2: Pääsy toisen saman luokan olion **private**-osaan

4.3 C++: const ja vakio-oliot

Monissa ohjelmointikielissä on jo ammoisista ajoista lähtien ollut mahdollisuus määritellä muuttujien lisäksi vakioita, jotka eroavat muuttujista siinä, että niiden arvoa ei voi muuttaa. C-kieleen tämä mahdollisuus tuli ANSI-standardin mukana 80-luvulla, kun kieleen lisättiin avainsana **const**. C++:ssa **const** on otettu mukaan myös olio-ominaisuuksiin, jossa sen käyttö on osoittautunut erittäin hyödylliseksi.

4.3.1 Perustyyppiset vakiot

C-kielessä perustyyppiset vakiot — lähinnä kokonaisluku- ja liukulukuvakiot — määriteltiin perinteisesti **#define**-esikääntäjäkomennolla. Syynä tähän oli, että vaikka ANSI-standardi toikin kieleen **const**-määreen, sen toiminta oli tehotonta eikä sitä voinut käyttää kaikissa tilanteissa. C++:ssa tilanne on korjattu, eikä **#define**-vakioita ole enää syytä käyttää.

Perustyyppinen vakio määritellään aivan kuten muuttuja, mutta tyypin yhteyteen lisätään määre **const**:

```
1   int const MAX_MJONON_KOKO = 30000;
2   double const PI = 3.14159265;
3   int const RAJA = annaRaja();
4   char mjono[MAX_MJONON_KOKO];
```

const-vakiot käyttäytyvät muuten kuin muuttujat, mutta niihin ei voi sijoittaa. Tämän lisäksi kokonaislukuvakiota voi käyttää myös paikoissa, joissa kääntäjä vaatii käännösaikaisia vakioita, kuten esimerkiksi taulukkojen ko'oissa (katso rivi 4 edellisessä esimerkissä).

C:stä poiketen vakioiden alustusarvon ei tarvitse olla käännösaikainen vaan se voi olla esim. funktion paluuarvo kuten rivillä 3 — tällaista ei-käännösaikaista vakiota ei kylläkään sitten voi käyttää esimerkiksi taulukon kokoa määräämään.

Kielen kannalta on aivan sama, onko sana **const** ennen tyypin nimeä vai sen jälkeen. Aiemmin käytettiin yksinomaan tapaa, jossa **const** tulee ennen tyyppiä (**const int** `i`), ja tätä tapaa näkee edelleenkin valtaosassa koodia. Tapa laittaa **const**-sana vasta tyypin nimen jälkeen on kuitenkin 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.) Tässä teoksessa on siirrytty käyttämään uutta käytäntöä vuoden 2005 painoksesta alkaen.

const-vakiot ovat normaalisti paikallisia siinä käännösyksikössä, jossa ne määritellään. Käytännössä tämä tarkoittaa sitä, että **const**-vakiot voi sijoittaa otsikkotiedostoihin, vaikka sinne ei normaalisti muuttujia laitetakaan.

Usein olio-ohjelmoinnissa vakiot liittyvät kiinteästi jonkin tietyn luokan käyttöön. Tällöin vakiot kannattaa kapseloida luokan sisään luokkavakioiksi, joista kerrotaan tarkemmin aliluvussa 8.2.2.

4.3.2 Vakio-oliot

Olioista voi tehdä “vakio-olioita” aivan samaan tapaan kuin perustyypeistä luodaan vakioita — lisätään olion määrittelyn jälkeen sana **const**:

```
Paivays const joulu(24,12,1999);
```

Olioiden tapauksessa **const**-sanan vaikutus on kuitenkin monimutkaisempi asia. Perustyypeistä puhuttaessa **const**-sanan vaikutus on helppo selittää — muuttujaan ei yksinkertaisesti saa sijoittaa. Olioiden tapauksessa tilanne on paljon hankalampi. Sijoittaminen ei ole välttämättä mielekäs toimenpide kaikille olioille (olioiden sijoittamista käsitellään aliluvussa 7.2). Jos halutaan puhua “vakio-olioista”, onkin oleellista se, ettei tällaisen vakio-olion tilaa pystytä muuttamaan. Koska olion tila on kapseloitu olion sisään, voi olion tila muuttua vain jäsenfunktiokutsun seurauksena.

C++:ssa vakio-olioiden käsite on toteutettu jakamalla jäsenfunktiot kahteen ryhmään — niihin, jotka voivat muuttaa olion tilaa ja niihin, jotka eivät. Tämän jälkeen on määrätty, että vakio-olioille saa kutsua vain niitä jäsenfunktioita, jotka eivät voi muuttaa olion tilaa. Saman asian voi myös ajatella niin, että C++:ssa vakio-olioilla on erilainen rajapinta, joka on vain osajoukko luokan koko rajapinnasta.

Jäsenfunktiot, joiden ei haluta muuttavan olion tilaa, merkitään määreellä **const**, joka tulee jäsenfunktion parametrilistan perään sekä luokan esittelyssä että jäsenfunktion määrittelyssä. Listauksessa 4.3 on luokan `Paivays` esittely, jossa päiväystä muuttamattomat jäsenfunktiot on merkitty vakioiksi. Listauksessa on myös esimerkkinä yhden jäsenfunktion määrittely.

Kääntäjä pitää huolen siitä, että **const**-sanan mukanaan tuomaa

```
 1 class Paivays
 2 {
 3 public:
 4    Paivays(unsigned int p, unsigned int k, unsigned int v);
 5    ~Paivays();
 6
 7    void asetaPaiva(unsigned int paiva);
 8    void asetaKk(unsigned int kuukausi);
 9    void asetaVuosi(unsigned int vuosi);
10
11    unsigned int annaPaiva() const;
12    unsigned int annaKk() const;
13    unsigned int annaVuosi() const;
14
15    void etene(int n);
16    int paljonkoEdella(Paivays const& p) const;
17
18 private:
19    unsigned int paiva_;
20    unsigned int kuukausi_;
21    unsigned int vuosi_;
22 };
```

määrittely

```
1 unsigned int Paivays::annaPaiva() const
2 {
3    return paiva_;
4 }
```

LISTAUS 4.3: Päiväysluokka **const**-sanoineen

"vakioisuutta" noudatetaan. Jo aiemmin on mainittu, että vakio-olioille voi kutsua vain vakiojäsenfunktioita. Tämän lisäksi kääntäjä pyrkii valvomaan, että vakiojäsenfunktioiden koodissa ei muuteta olion tilaa. Tämän se tekee asettamalla seuraavat rajoitukset vakiojäsenfunktion koodille:

- Vakiojäsenfunktion koodissa ei voi muuttaa jäsenmuuttujien arvoja (toisin sanoen jäsenmuuttujat käyttäytyvät ikään kuin ne olisi määritelty **const**-määreellä).
- Vakiojäsenfunktion koodissa voi kutsua omalle oliolle vain toisia vakiojäsenfunktioita. Tämä rajoitus koskee siis vain jäsenfunktion omaan olioon kohdistuvia kutsuja.
- Vakiojäsenfunktion koodissa osoitin **this** on tyyppiä "osoitin vakio-olioon" (katso seuraava aliluku).

Edellä mainitut rajoitukset eivät kuitenkaan riitä varmistamaan täydellisesti, ettei vakiojäsenfunktiossa olion tila muutu. Osa olion tilaan kuuluvasta tiedostahan saattaa nimittäin olla olion ulkopuolella esim. osoittimien päässä. Tällaiseen dataan eivät kääntäjän tarkastukset ulotu, vaan ohjelmoijan on itsensä pidettävä huoli siitä, ettei vakiojäsenfunktiossa muuteta mitään sellaista, jonka katsotaan kuuluvan olion tilaan.

Joskus harvoin saattaa olla aihetta määritellä jäsenmuuttuja, jota voisi muuttaa myös vakiojäsenfunktioissa. Tällainen on perusteltua vain, jos jäsenmuuttujan muuttaminen ei vaikuta olion "todelliseen" tilaan. Tällaisia jäsenmuuttujia on mahdollista saada aikaan lisäämällä niiden esittelyn eteen avainsana **mutable**.

C++ antaa myös mahdollisuuden siihen, että luokka tarjoaa kaksi samannimistä jäsenfunktiota *samoilla parametreilla*, jos toinen on vakiojäsenfunktio ja toinen ei. Tällaisessa tapauksessa jäsenfunktion kutsuminen toimii niin, että vakio-olioille ja vakio-osoittimien ja -viitteiden läpi kutsutaan vakiojäsenfunktiota, muuten tavallista. Tästä erottelusta on joskus hyötyä, koska vakiojäsenfunktio voi esimerkiksi palauttaa vakio-osoittimen olion dataan, tavallinen jäsenfunktio taas normaalin osoittimen.

4.3.3 Vakioviitteet ja -osoittimet

Varsinaisia vakio-olioita tarvitaan olio-ohjelmoinnissa äärimmäisen harvoin, koska olio-ohjelmoinnin yksi perusajatuksista on, että olioiden tila muuttuu niihin kohdistettujen toimintojen tuloksena. Vakio-olion käsite muuttuu kuitenkin erittäin käyttökelpoiseksi, kun otetaan käyttöön vakioviitteet ja -osoittimet.

Vakioviitteellä tarkoitetaan tässä kirjassa viitettä, jonka *läpi* asiat näyttävät vakioilta. Vastaavasti vakio-osoitin on osoitin, jota käytettäessä sen päässä oleva asia vaikuttaa vakiolta. Termiä "vakio-osoitin" ei tule sekoittaa termiin "osoitinvakio", jolla tarkoitetaan osoitinta, joka *itse* on vakio, ts. osoitinta ei voi muuttaa osoittamaan toiseen paikkaan. Huomaa, että kaikki alla esitetyt asiat pätevät niin vakio-osoittimille kuin vakioviitteille, vaikka tekstissä mainittaisiinkin vain toinen.

Osoittimista ja viitteistä saadaan vakio-osoittimia ja -viitteitä lisäämällä niiden esittelyyn määre **const**:

```
char const* mjono;
Paivays const& p;
```

Vakio-osoittimia käytettäessä osoittimen päässä oleva olio tai data käyttäytyy *ikään kuin* se olisi vakio — riippumatta siitä, onko olio tai data alunperin määritelty vakioksi. Tämä siis tarkoittaa sitä, että vakio-osoittimen läpi sijoittaminen ja muiden kuin vakiojäsenfunktioiden kutsuminen on mahdotonta.

Vakioviitteiden hyöty tulee siitä, että niiden avulla voidaan oliosta näkyvä rajapinta rajata sellaiseksi, että olion muuttaminen vakioviitteen kautta tulee mahdottomaksi. Jos esimerkiksi funktio ottaa parametrikseen vakioviitteen päiväysolioon, voi funktion käyttäjä luottaa siihen, että funktiolle välitetyn päiväysolion sisältämä päiväys on varmasti sama myös funktiokutsun jälkeen. Vakioviitteiden käyttö on myös tehokas "dokumentointikeino", jolla voi kertoa, ettei tiettyä oliota tai dataa ole tarkoitus muuttaa. Erityisen tehokkaaksi tämän dokumentointikeinon tekee se, että kääntäjä takaa sen noudattamisen. Esimerkiksi seuraava koodi ei mene kääntäjästä läpi:

```
void muutanKuitenkin(Paivays const& pvm)
{
  pvm.asetaPaiva(1); // KÄÄNNÖSVIRHE: asetaPaiva ei vakiojf.
}
```

C++ sisältää automaattiset tyyppimuunnokset ei-vakio-osoittimista vakio-osoittimiksi (ja sama viitteille), mutta ei toiseen suuntaan:

```
1  char* mjono = "Käytä string-luokkaa char*:n sijaan";
2  char const* vakiomjono = mjono; // Ok: ei-vakio ⇒ vakio
3  char* mjono2 = vakiomjono; // KÄÄNNÖSVIRHE: vakio ⇒ ei-vakio
```

Nämä kielen säännöt tarkoittavat käytännössä, että ei-vakio-olioitakin voi käsitellä ikään kuin ne olisivat vakioita, mutta vakio-olioista ei millään saa tavallisia. Tämä on järkevää, kun muistetaan, että vakio-olion rajapinta on vain osajoukko normaaliolion rajapinnasta.

Vakioviitteiden yleisin käyttökohde on epäilemättä funktioiden ja jäsenfunktioiden parametrit. Mikäli funktio ottaa parametrinaan viitteen olioon, jota sen ei ole tarve muuttaa, tulisi parametriviitteen olla ***aina*** vakioviite. Sama pätee tietysti myös osoittimille. Osoittimia käytetään yleisesti myös olioiden jäsenmuuttujina yms. Tällöin kannattaa miettiä, onko osoittimen läpi tarpeen muuttaa oliota. Jos vastaus on ei, kannattaa käyttää vakio-osoitinta.

Edellä on jo mainittu kaksi vakioviitteiden ja -osoittimien tärkeää käyttösyytä: käyttö dokumentointimielessä rajapintaa rajaamaan ja kääntäjän tekemät tarkastukset siitä, että vakiorajapintaa todella noudatetaan. Kolmas syy käyttää vakioviitteitä esimerkiksi funktioiden parametrina on niin yksinkertainen, että se unohtuu helposti: vakio-olion voi laittaa *ainoastaan* vakioviitteen tai vakio-osoittimen päähän.

Jos funktio ottaa parametrinaan tavallisen viitteen olioon, ei funktiolle voi antaa parametrina vakio-oliota, koska tavallisen viitteen kautta tulisi olion muuttaminen mahdolliseksi. Listauksessa 4.4 seuraavalla sivulla on esimerkki tyypillisestä tilanteesta, joka syntyy kun yhdestä funktiosta unohtuu **const**-sana viitteen edestä pois. Funktio `saakoHameenKuukaudessa`[†] on kirjoitettu oikeaoppisesti niin, että se ottaa parametrinaan vakioviitteen päiväykseen — eihän funktion ole tarkoitus muuttaa parametrina tullutta päiväystä. Funktion toteutuksessa kuitenkin kutsutaan toista funktiota `onkoKarkauspaivaa`, jonka kirjoittaja ei ole käyttänyt vakioviitettä parametrina, vaikka karkauspäivän testaamisen ei varmaankaan ole tarkoitus muuttaa testattavaa

[†] Funktio testaa, onko annetusta päivästä kuukauden sisällä mahdollisuus saada hamekangasta. (Karkauspäivänä kosimisesta kieltäytyvän täytyy ostaa kosijalle hamekangas. Epäreilua niitä miehiä kohtaan, jotka eivät käytä hametta.)

päivää. Tämä aiheuttaa käännösvirheen, kun vakioviitettä yritetään antaa parametrina funktiolle, jonka ottama viite ei ole vakio.

Toinen tyypillinen vakio-olioihin liittyvä virhe on, että luokkaa suunniteltaessa unohdetaan varustaa olion tilaa muuttamattomat jäsenfunktiot **const**-määreellä. Tällöin käy niin, että vakioviitteiden kautta oliolle ei voi kutsua ainuttakaan jäsenfunktiota! Aiemmin tässä luvussa ollut `PieniPaivays`-luokka (listaus 2.1 sivulla 62) on esimerkki tällaisesta virheellisestä luokasta, jossa luokan suunnittelijan hutilointi estää luokan käyttäjiä hyödyntämästä kielen turvaominaisuuksia. Tällaisten virhetilanteiden varalle C++-kielessä on tyyppimuunnos **const_cast**, josta kerrotaan aliluvussa 7.4.1.

Joskus hyvin harvoin on tarve saada itse osoittimesta vakio sen osoittaman olion sijaan — toisin sanoen halutaan, että osoitinta ei voi muuttaa osoittamaan toiseen paikkaan. Silloin puhutaan *osoitinvakiosta*. C++:n syntaksi seuraa tässä kohtaa tämän kirjan käytäntöä laittaa **const** aina tyypin jälkeen. Osoittimen saa vakioksi lisäämällä **const**-sanan osoitintyypin tähden jälkeen:

```
char* const osoitinvakio = "Loogista, eikö totta";
```

Vastaavasti osoittimen, jota ei saa muuttaa ja jonka osoittamaa dataa ei myöskään saa muuttaa, tyyppi on **char const* const**.

```
1  #include "paivays.hh"
2
3  bool onkoKarkauspaivaa(Paivays* pvm_p);
4
5  bool saakoHameenKuukaudessa(Paivays const& nyt)
6  { // Karkauspäivä on kuukauden sisällä, jos on helmikuu ja
7    // vuoden sisälle osuu karkauspäivä
8    if (nyt.annaKk() != 2)
9    {
10     return false; // Ei helmikuu
11   }
12   else
13   {
14     return onkoKarkauspaivaa(&nyt); // KÄÄNNÖSVIRHE
15   }
16 }
```

LISTAUS 4.4: Esimerkki virheestä, kun **const**-sana unohtuu