Osa II

Peruskieli

Kirjoittamamme ohjelmakoodin teksti ja käsittelemämme tieto on tallennettu tietokoneen muistiin bittijonoina. *Bitti* on yksittäinen solu, jolla voi olla arvo 0 tai 1. Fysiikan termein tämä arvo on sähköinen varaus, joka on joko “pois päältä” tai “päällä”. Tyypillinen tietokoneen muistisegmentti voi näyttää tältä:

```
00011011011100010110010000111011...
```

Tämä bittikokoelma tällä tasolla on ilman rakennetta. On vaikeaa puhua tästä bittijonosta millään mielekkäällä tavalla. Silloin tällöin on kuitenkin välttämätöntä tai kätevämpää ohjelmoida yksittäisten tai ryhmäkokonaisuuksia muodostavien bittien tasolla (etenkin, kun käsittelemme itse koneen laitteistoa). C++-kielessä on joukko bittioperaattoreita, jotka tukevat bittikäsittelyä, ja bittijoukolle on säiliötyyppi, jolla voidaan esitellä bittikokoelmia olioille (näitä operaattoreita ja bittijoukkosäiliöitä käsitellään luvussa 4).

Rakenne määräytyy bittijonosta ajattelemalla bittejä ryhminä, joita kutsutaan *tavuiksi* ja *sanoiksi*. Yleensä tavu muodostuu 8 bitistä. Sana muodostuu tyypillisesti 32 bitistä eli 4 tavusta (työasemien käyttöjärjestelmät ovat kuitenkin nykyään siirtymässä 64-bittisiin järjestelmiin). Sanojen koko voi vaihdella koneesta toiseen. Puhumme näiden arvojen olevan *koneriippuvaisia* (*machine-dependent*). Seuraava kuva esittää bittivirtaamme järjestettynä neljään osoitettavissa olevaan riviin.

1024	0	0	0	1	1	0	1	1
1032	0	1	1	1	0	0	0	1
1040	0	1	1	0	0	1	0	0
1048	0	0	1	1	1	0	1	1

Osoitettavissa oleva koneen muisti

Muistin järjestely mahdollistaa, että voimme viitata tiettyyn bittikokoelmaan. Siksi on mahdollista puhua sanasta osoitteessa 1024 tai tavusta osoitteessa 1040, ja voimme sanoa esimerkiksi, että tavu osoitteessa 1032 ei ole yhtä suuri kuin tavu osoitteessa 1048.

Kuitenkaan ei ole mahdollista puhua mielekkäästi tavun sisällöstä osoitteessa 1032. Miksi? Koska emme tiedä, kuinka tulkitsisimme sen bittijonon. Puhuaksemme mielekkäästi tavusta osoitteessa 1032, meidän pitää tietää esitettävän arvon tyyppi.

Tyyppiabstraktio mahdollistaa mielekkään tulkinnan kiinteämittaisista bittijonoista. C++:ssa on esimääritelty joukko tietotyyppejä kuten merkit, kokonaisluvut ja liukulukunumerot ja joukko perustietoabstraktioita kuten merkkijono, vektori ja kompleksinumerot. Siinä on myös joukko operaattoreita, kuten lisäys-, vähennys-, yhtäsuuruus- ja pienempi kuin -operaattorit näiden tyyppien käsittelemiseksi ja pieni joukko lauseita ohjelmankulun kontrolloimiseksi, kuten while-silmukka ja if-lause. Nämä elementit muodostavat aakkoset, joilla monet laajat ja monimutkaiset tosielämän järjestelmät on kirjoitettu. Ensimmäinen vaihe C++:n hallitsemisessa on näiden peruskomponenttien ymmärtäminen, joka muodostaa tämän *C++*-kirjan II-osan aihealueen.

Luvussa 3 silmäillään esimääriteltyjä ja laajennettuja tietotyyppejä sekä katsotaan uusien tyyppien rakentamismekanismeja, pääosin luokkamekanismia, joka esiteltiin kohdassa 2.3. Luvussa 4 keskitytään lauseketukeen, katsellaan esimääriteltyjä operaattoreita sekä käsitellään myös aiheita tyyppikonversiosta, operaattorin sidontajärjestyksestä ja assosiatiivisuudesta. Ohjelmiemme pienin yksikkö on lause. Ohjelmalauseet ovat luvun 5 aiheena. Luvussa 6 esitellään vakiokirjaston säiliötyypit kuten vektori ja assosiatiivinen taulukko ja niiden käyttöä tekstinkyselyjärjestelmän toteutuksen kautta.

3

C++:n tietotyypit

Tässä luvussa silmäillään *sisäänrakennettuja* eli *primitiivisiä* tietotyyppejä, jotka on esimääritelty C++:ssa. Luku alkaa katsauksella *literaalivakioihin* kuten 3.14159 ja "pii" ja esittelee käsitteen *symbolinen muuttuja* eli *olio*. C++-ohjelman olio pitää määritellä tietyn tyyppiseksi. Luvun loppuosassa käydään läpi eri tyypit, jollaisiksi oliot voidaan esitellä. Lisäksi vertailemme C++:n sisäistä merkkijono- ja taulukkotukea luokka-abstraktioihin, jotka on toteutettu C++-vakiokirjastoon. Vaikka kirjastoabstraktiot eivät ole primitiivisiä tyyppejä, ne ovat perustyyppejä tosielämän C++-ohjelmissa, ja haluamme esitellä ne niin pian kuin mahdollista, jotta voimme rohkaista ja esitellä niiden käyttöä. Haluamme viitata näihin tyyppeihin *laajennettuna peruskielenä*, joka muodostuu primitiivisistä sisäisistä tyypeistä ja primitiivisistä luokka-abstraktio-tyypeistä.

3.1 Literaalivakio

C++:ssa on joukko esimääriteltyjä numeerisia tietotyyppejä, joilla voidaan esittää kokonaislukuja, liukulukuja ja yksittäisiä merkkejä, sekä esimääritelty merkkitaulukko, jolla voidaan esittää merkkijonoja.

- Tyyppiä char käytetään tyypillisesti esitettäessä yksittäisiä merkkejä ja pieniä kokonaislukuja. Se esitetään yksittäisessä koneen tavussa.
- Tyypit short, int ja long edustavat erikokoisia kokonaislukuarvoja. Tyypillisesti short-tyypit esitetään koneen puolisanassa, int-tyypit koneen sanassa ja long-tyypit joko yhdessä tai kahdessa koneen sanassa (32-bittisissä koneissa int- ja long-tyypit ovat usein samanpituisia).
- Tyypit float, double ja long double edustavat liukuluvun yksinkertaisen, kaksois- ja laajennetun tarkkuuden arvoja. Tyypillisesti float-tyypit esitetään yhdessä sanassa, double-tyypit kahdessa sanassa ja long double joko kolmessa tai neljässä sanassa.

char-, short-, int- ja long-tyyppejä kutsutaan yhteisellä nimellä *kokonaistyypeiksi* (*integral*

types). Kokonaistyypit voivat olla joko etumerkillisiä tai etumerkittömiä. Etumerkillisellä tyypillä vasemmanpuoleisin bitti toimii *etumerkkibittinä* ja muut bitit edustavat arvoa. Etumerkittömällä tyypillä kaikki bitit edustavat arvoa. Jos etumerkkibitin arvo on 1, arvo tulkitaan negatiiviseksi; jos se on 0, arvo on positiivinen. 8-bittinen signed char voi sisältää arvoja väliltä –128 ... 127 ja etumerkitön char väliltä 0 ... 255.

Ohjelmassa esiintyvää arvo kuten 1 kutsutaan sitä *literaalivakioksi*: literaali siksi, koska voimme puhua siitä vain sen arvon ehdoilla, ja vakio siksi, koska sen arvoa ei voi muuttaa. Jokaisella literaalilla on siihen liittyvä tyyppi. Esimerkiksi 0 on int-tyyppiä. Arvo 3.14159 on double-tyyppinen literaalivakio. Pidämme literaalivakiota *osoitteettomana*; vaikka sen arvo on talletettu jonnekin tietokoneen muistiin, ei meillä ole keinoja käsitellä tuota osoitetta.

Kokonaislukujen literaalit voidaan kirjoittaa desimaalisella-, oktaalisella- tai heksadesimaalisella merkintätavalla. (Tämä ei muuta arvon bittiesitystapaa.) Esimerkiksi arvo 20 voidaan kirjoittaa millä tahansa seuraavista kolmesta tavasta:

```
20   // desimaalinen
024  // oktaalinen
0x14 // heksadesimaalinen
```

Kun kokonaisluvun literaalin eteen lisätään 0 (nolla), se tulkitaan oktaaliseksi. Kun kokonaisluvun literaalin eteen lisätään joko 0x tai 0X, se tulkitaan heksadesimaaliseksi. (Luvussa 20, Iostream-kirjasto, käsitellään arvojen tulostamista oktaalisella- tai heksadesimaalisella merkintätavalla.)

Oletusarvoisesti kokonaislukujen literaalivakioita kohdellaan int-tyypin etumerkillisinä arvoina. Kokonaisluvun literaalivakio voidaan määrittää long-tyyppiseksi laittamalla arvon perään L tai l (kirjain “äl”, joko pienellä tai isolla). Pienen kirjaimen käyttöä tulisi välttää, koska se ymmärretään helposti väärin numeroksi 1. Samalla tavalla kokonaisluvun literaalivakio voidaan määrittää etumerkittömäksi laittamalla arvon perään joko U tai u. Etumerkitön long-literaalivakio voidaan myös määrittää. Esimerkiksi:

```
128u    1024UL   1L   8Lu
```

Liukuluvun literaalivakio voidaan kirjoittaa joko tieteellisellä tai yleisellä desimaalisella merkintätavalla. Kun käytetään tieteellistä merkintätapaa, eksponentti voidaan kirjoittaa joko E tai e. Oletusarvoisesti liukuluvun literaalivakiota kohdellaan double-tyyppisenä. Yksinkertaisen tarkkuuden literaalivakio ilmaistaan laittamalla arvon perään joko F tai f. Samalla tavalla laajennettu tarkkuus ilmaistaan laittamalla arvon perään joko L tai l (jälleen kannatetaan ison kirjaimen käyttöä). (Huomaa myös, että jälkiliitteitä F, f, L ja l voidaan käyttää vain, kun käytetään yleistä desimaalista merkintätapaa.) Esimerkiksi:

```
3.14159F 0.1f   12.345L  0.0
3e1      1.0E-3 2.       1.0L
```

Sanat true ja false ovat bool-tyypin literaaleja. Voidaan kirjoittaa esimerkiksi:

```
true        false
```

Tulostettavissa olevan merkin literaalivakio voidaan kirjoittaa sulkemalla merkki puoli-

lainausmerkkeihin. Esimerkiksi:

```
'a'     '2'     ','     ' ' (tyhjä merkki)
```

Valikoidut tulostumattomat merkit, puolilainaus- ja lainausmerkit sekä kenoviiva voidaan esittää seuraavilla *koodinvaihtomerkeillä* (*escape sequences*) (koodinvaihto alkaa kenoviivalla):

rivinvaihto	\n
vaakasarkain	\t
pystysarkain	\v
askelpalautin	\b
rivinalkuun	\r
sivunvaihto	\f
hälytys (kello)	\a
kenoviiva	\\
kysymysmerkki	\?
puolilainausmerkki	\'
lainausmerkki	\"

Yleistetyllä koodinvaihtomerkillä on muoto

```
\ooo
```

jossa ooo edustaa kolmea oktaalinumeroa. Oktaalinumeron arvo edustaa merkin numeerista arvoa koneen merkistössä. Seuraavat esimerkit esittävät literaalivakioita, jotka käyttävät ASCII-merkistöä:

```
\7 (kello)          \14 (rivinvaihto)
\0 (null)           \062 ('2')
```

Lisäksi merkkiliteraalin eteen voidaan laittaa L kuten seuraavassa:

```
L'a'
```

Tätä kutsutaan *leveän merkin literaaliksi* ja sen tyyppi on wchar_t. Leveiden merkkien literaalit tukevat kielimerkistöjä kuten Kiinassa ja Japanissa, joissa joitakin merkkiarvoja ei voida esittää yhdellä char-merkillä.

Merkkijonon literaalivakio muodostuu nollasta tai useammasta merkistä, jotka ovat lainausmerkkien sisällä. Tulostumattomat merkit voidaan esittää vastaavilla koodinvaihtomerkeillä. Merkkijonoliteraali voi jakautua useammalle riville. Kenoviiva rivin viimeisenä merkkinä ilmaisee, että merkkijonoliteraali jatkuu seuraavalle riville. Esimerkiksi:

```
"" (tyhjä merkkijono)
"a"
"\nCC\toptions\tfile.[cC]\n"
"monirivinen \
merkkijonoliteraali ilmaisee \
jatkumisestaan takakenolla"
```

Merkkijonoliteraali on tyyppiä *vakiomerkkien taulukko*. Se muodostuu sekä merkkijonoliteraalista että sen päättävästä null-merkistä, jonka kääntäjä lisää. Esimerkiksi siinä, kun

```
'A'
```

edustaa yhtä merkkiä A, seuraava edustaa yhtä merkkiä A, jonka perässä on null-merkki:

```
"A"
```

Null-merkki on C- ja C++-kielen tapa ilmaista merkkijonon loppu.

Samoin kuin on olemassa leveän merkin literaali, kuten

```
L'a'
```

on olemassa leveä merkkijonoliteraali ja jälleen sen edessä L, kuten seuraavassa:

```
L"leveän merkkijonon literaali"
```

Leveän merkkijonon literaali on tyyppiä *leveiden vakiomerkkien taulukko*. Se päätetään myös vastaavalla leveällä null-merkillä.

Jos kaksi merkkijonoa tai leveää merkkijonoa on ohjelmassa vierekkäin, ne yhdistetään ja loppuun laitetaan null-merkki. Esimerkiksi

```
"two" "some"
```

tulostuu yhtenä sanana *twosome*. Mitä tapahtuu, jos yrität yhdistää merkkijonoliteraalin ja leveän merkkijonoliteraalin? Esimerkiksi:

```
// tämä ei ole hyvä ajatus
"two" L"some"
```

Tulos on *tuntematon* — tämä tarkoittaa, että ei ole määritelty vakiokäytäntöä kahden eri tyypin yhdistämiselle. Ohjelmista, joissa käytetään vastaavaa tuntematonta käyttäytymistä, sanotaan, että ne eivät ole *siirrettäviä* (*nonportable*). Vaikka ohjelma voi toimia oikein nykyisellä kääntäjällä, ei ole takuita siitä, että sama ohjelma toimisi oikein eri kääntäjällä tai saman kääntäjän myöhemmällä versiolla. Tämänkaltaisten ongelmien jäljittäminen aikaisemmin toimivista ohjelmista on lievästi sanottuna epämiellyttävä tehtävä. Suosittelemme, että ei kannata tietoisesti hyödyntää tuntemattomien ohjelmien idiomeja. Osoitamme sopivissa kohdissa sellaiset idiomit.

Harjoitus 3.1

Selitä seuraavien literaalivakioiden väliset erot:

(a) 'a', L'a', "a", L"a"
(b) 10, 10u, 10L, 10uL, 012, 0xC
(c) 3.14, 3.14f, 3.14L

Harjoitus 3.2

Mitkä seuraavista eivät ole sallittuja, vai ovatko kaikki sallittuja?

(a) "Who goes with F\144rgus?\014"
(b) 3.14e1L
(c) "two" L"some"
(d) 1024f
(e) 3.14UL
(f) "multiple line
comment"

3.2 Muuttujat

Kuvitellaan, että meille on annettu tehtävä laskea 2 potenssiin 10. Ensimmäinen yrityksemme voisi näyttää tältä:

```
#include <iostream>

int main() {
   // ensimmäinen ratkaisu
   cout << "2 korotettuna potenssiin 10: ";
   cout << 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2;
   cout << endl;
   return 0;
}
```

Tämä ratkaisee ongelman, vaikka saatamme tarkistaa kahdesti tai kolmesti, että literaalista 2 on täsmälleen 10 esiintymää kerrottuna yhteen. Muutoin olemme tyytyväisiä. Ohjelmamme generoi vastauksen 1024 oikein.

Sitten meitä pyydetään laskemaan 2 korotettuna potenssiin 17 ja sitten potenssiin 23. Ohjelmamme muuttaminen on harmillista joka kerta. Pahempi asia on, että se osoittautuu merkittävän virhealttiiksi. Liian usein muokattu ohjelma tuottaa vastauksen, jossa on liian vähän tai liian paljon 2:n esiintymiä. Lopuksi meitä pyydetään tuottamaan taulukkomuotoinen listaus, jossa on 2 korotettuna potenssiin väliltä 0 — 15. Literaalivakioiden käyttö vaatii 32 riviä tavallisessa muodossa:

```
cout << "2 korotettuna potenssiin X\t";
cout << 2 * ... * 2;
```

jossa X kasvaa arvolla 1 joka lauseparissa.

Toisaalta tällä saadaan työ tehdyksi; työnantajamme tuskin kaivaa esille, kuinka olemme toteuttaneet työn, edellyttäen, että tulokset ovat oikein ja tulevat ajallaan. Itse asiassa monissa tuotantoympäristöissä menestyksen pääasiallisin mittari on lopullinen tulostus ja kaikkea puhetta prosessista pidetään todennäköisesti akateemisena ja toissijaisena.

Vaikka “raaka voima” ratkaisuna toimii, on prosessi usein epämiellyttävä ja kriisin kajon sävyttämä. Viehtymys sellaisiin ratkaisuihin johtuu niiden yksinkertaisuudesta. Ymmärrämme

tarkasti, mitä pitää tehdä, vaikka se ei ole yleensä hauskaa. Teknisesti hienostuneemmat ratkaisut vaativat yleensä melkoisesti enemmän aikaa alkuvaiheessa ja silloin tuntuu, että mitään ei ole vielä saatu toteutetuksi. Ja koska prosessi on automatisoitu, on olemassa suuremmat mahdollisuudet, että asiat menevät pieleen.

Ja asiat väistämättä näyttävät menevän pieleen. Hyöty on, että väistämättömien virheiden ja harha-askelten keskellä ei ainoastaan tehdä asioita nopeammin, vaan myös mahdollisuuksien rajoja laajennetaan. Joskus, ihme kyllä, prosessista tulee hauskempaa.

Esimerkkimme selkeän työlääälle 2:n potenssilaskulle on olemassa kaksikin vaihtoehtoa: sellaisten nimettyjen olioiden käyttö, jotka lukevat ja kirjoittavat vaiheittaiset laskennat ja ohjelmankulun ohjausrakenteet ohjelmalauseiden toistuvalle suorittamiselle niin kauan, kun jokin totuusarvo on tosi. Tässä sitten on vaihtoehtoinen “teknisesti kehittynyt” 2:n korotus potenssiin 10:

```
#include <iostream>

int main()
{
        // oliot tyyppiä int
        int value = 2;
        int pow = 10;

        cout << value << " korotettuna potenssiin "
            << pow << ": \t";

        int res = 1; // tulee sisältämään tuloksen

        // silmukan ohjauslauseet: toista res-laskentaa,
        // kunnes cnt on suurempi kuin pow
        for ( int cnt=1; cnt <= pow; ++cnt)
                res = res * value;

        cout << res << endl;
}
```

value, pow, res ja cnt ovat muuttujia, jotka mahdollistavat niiden arvojen tallentamisen, muokkaamisen ja hakemisen. Laskenta voidaan toistaa for-silmukalla, kunnes se on suoritettu pow kertaa.

Vaikka tämän tasoinen yleistys tekee ohjelmasta joustavamman, ei se ole vielä uudelleenkäytettävä ohjelma. Yleistämisen tason nosto on välttämätöntä: otetaan ohjelmasta erilleen osa, joka laskee potenssiinkorotuksen arvon, ja määritellään se itsenäiseksi funktioksi, jonka muut voivat käynnistää. Esimerkiksi:

```
int
pow( int val, int exp )
{
   for ( int res = 1; exp > 0; --exp )
        res = res * val;
```

```
    return res;
}
```

Nyt jokainen ohjelmamme, jonka pitää laskea potenssiinkorotuksen arvo, voi käyttää tätä pow()-ilmentymää sen sijaan, että toteuttaisi sen uudelleen itse. 2:n potenssin taulukon generointi voidaan tehdä seuraavasti:

```
#include <iostream>
extern int pow(int,int);

int main()
{
    int val = 2;
    int exp = 15;

    cout << "2:n potenssi\n";
    for ( int cnt=0; cnt <= exp; ++cnt )
            cout << cnt << ": "
                << pow(val,cnt) << endl;

    return 0;
}
```

Käytännössä toteutuksemme pow()-funktiosta ei ole vankka eikä tarpeeksi yleinen. Mitä jos eksponentti on negatiivinen, tai jos se on esimerkiksi 1 000 000? Negatiiviselle eksponentille funktiomme palauttaa aina arvon 1. Erittäin suurille eksponenteille int res -muuttujamme on liian pieni, jotta se voisi sisältää tuloksen arvon. Sen sijaan erittäin suurille eksponenteille palautetaan mielivaltainen ja virheellinen arvo. (Paras ratkaisu tässä tapauksessa on muokata toteutustamme ja laittaa palautustyypiksi double-tyyppi.) Yleisyyden nimissä funktiomme tulisi kyetä käsittelemään kokonaislukujen ja liukulukujen arvoja, eksponentteja jne. Kuten näet, vahvan ja yleisen funktion kirjoittaminen tuntemattomalle käyttäjäjoukolle on monimutkaisempaa kuin tietyn algoritmin toteutus, joka vastaa välitöntä tarvettamme. (Jos haluat nähdä tosielämän toteutuksen pow()-funktiosta, katso julkaisusta [PLAUGER92].)

3.2.1 Mikä muuttuja on?

Muuttuja on nimetty muistialue, johon voimme kirjoittaa, hakea ja käsitellä läpi koko ohjelmankulun. Jokaiseen symboliseen muuttujaan liittyy C++:ssa tietty tyyppi, joka määrää siihen liittyvän muistin koon ja asettelun, talletettavan arvoalueen ja joukon operaatioita, joita siihen voidaan käyttää. Puhumme muuttujasta vaihtoehtoisesti *oliona*. Esimerkiksi tässä on viiden erilaisen olion määrittelyt (katsomme määrittelyn yksityiskohdat seuraavassa kohdassa):

```
int    student_count;
double salary;
bool   on_loan;
string street_address;
```

```
char   delimiter;
```

Sekä muuttuja että literaalivakio vievät muistia ja niillä on vastaavat tyypit. Ero on, että muuttuja on *osoitettavissa* (*addressable*). Muuttujaan liittyy kaksi arvoa:

1. Sen tiedon arvo, joka on tallennettu jonnekin muistiosoitteeseen. Tätä nimitetään joskus olion *rvalue*ksi (lausutaan englanniksi “are-value”). Voit ajatella, että rvalue tarkoittaa *luettavaa arvoa* (*read value*). Sekä literaalivakio että muuttuja voivat toimia rvaluena.
2. Sen osoitteen arvo — tarkoittaa sen osoitetta muistissa, johon sen tiedon arvo on tallennettu. Tätä nimitetään joskus muuttujan *lvalue*ksi (lausutaan englanniksi “ell-value”). Voit ajatella, että lvalue tarkoittaa *sijaintipaikan arvoa* (*location value*). Literaalivakio ei voi toimia lvaluena.

Lausekkeessa

```
ch = ch - '0';
```

ch-muuttuja esiintyy sijoitusoperaattorin sekä oikealla että vasemmalla puolella. Oikeanpuoleinen ilmentymä luetaan. Siihen liittyvä tieto haetaan sen osoitteen mukaisesta muistipaikasta. Vasemmanpuoleiseen esiintymään kirjoitetaan. Vähennysoperaation tulos tallennetaan ch:n sijaintipaikan arvoon; sen aikaisemman arvon päälle kirjoitetaan. Lausekkeen oikeanpuoleiset ch ja merkin literaalivakio toimivat rvalueina. Lausekkeen oikeanpuoleinen ch toimii lvaluena.

Yleensä sijoitusoperaattorin vasemmalle puolelle vaaditaan lvalue. Tapahtuu esimerkiksi käännöksenaikainen virhe, jos kirjoitetaan kumpi tahansa seuraavista:

```
// käännöksenaikaiset virheet: vasen puoli ei ole lvalue

// virhe: literaalivakio ei ole lvalue
0 = 1;

// virhe: aritmeettinen lauseke ei ole lvalue
salary + salary * 0.10 = new_salary;
```

Tässä tekstissä näemme lukuisia tilanteita, joissa rvaluen tai lvaluen käyttö vaikuttaa ohjelmiemme käyttäytymiseen ja suorituskykyyn — etenkin, kun välitetään ja palautetaan arvoja funktiosta.

Muuttujan määrittely saa aikaan, että siihen liittyvä muistialue varataan. Koska oliolla voi olla vain yksi sijaintipaikka, voi olla vain yksi määrittely jokaisella ohjelmamme oliolla. Tämä voi olla ongelma, jos olio on määritelty yhdessä tiedostossa, mutta sitä pitää käsitellä toisessa. Esimerkiksi:

```
// tiedosto module0.C
// määrittelee olion fileName
string fileName;

// ... sijoita fileName-olioon arvo
```

```
// tiedosto module1.C
// tarve käyttää fileName-oliota

// hups: käännös epäonnistuu:
// fileName on tuntematon module1.C:ssä
ifstream input_file( fileName );
```

C++:ssa pitää olio tehdä tunnetuksi ohjelmalle ennen sen käyttöä. Tämä on välttämätöntä, jotta kääntäjä voisi taata tyypin oikeellisuuden olion käytön yhteydessä. Viittaaminen tuntemattomaan olioon johtaa käännöksenaikaiseen ohjelmavirheeseen. Esimerkissämme module1.C:n käännös epäonnistuu, koska fileName on tuntematon sen sisällä.

Jotta module1.C voitaisiin kääntää, pitää fileName tehdä tunnetuksi ohjelmalle, mutta tekemättä toista määrittelyä. Teemme sen *esittelemällä* (*declare*) muuttujan:

```
// tiedosto module1.C
// haluaa käyttää fileName-oliota

// esittelee muuttujan fileName, joka tarkoittaa, että tekee siitä tunnetun
// ohjelmalle tekemättä toista määrittelyä
extern string fileName;

ifstream input_file( fileName );
```

Olion *esittely* saa sen nimen ja tyypin tunnetuksi ohjelmalle. Se muodostuu olion nimestä, tyypistä ja edessä olevasta avainsanasta extern. (extern-sanan täydellinen käsittely kohdassa 8.2.) Esittely ei ole määrittely eikä johda muistitilan varaamiseen. Itse asiassa se on väittämä, että muuttujan määrittely on jossain muualla ohjelmassa.

Vaikka ohjelma voi sisältää oliolle vain yhden määrittelyn, se voi sisältää vaikka kuinka monta olion esittelyä. Sen sijaan, että tehtäisiin erillinen esittely jokaiseen tiedostoon, jossa oliota käytetään, on parempi esitellä olio otsikkotiedostossa ja ottaa se mukaan aina, kun esittelyä tarvitaan. Tällä tavalla, jos esittelyä pitää muokata, teemme sen vain kerran ja saavutamme yhtenäisen esittelyn monissa tiedostoissa, jotka käyttävät tuota esittelyä. (Meillä on lisää sanottavaa otsikkotiedostoista kohdassa 8.2.)

3.2.2 Muuttujan nimi

Muuttujan nimi, sen *tunnus* (*identifier*), voi muodostua kirjaimista, numeroista ja alaviivamerkistä. Sen pitää alkaa joko kirjaimella tai alaviivalla. Isot ja pienet kirjaimet eroavat toisistaan. Kieli ei rajoita nimen sallittua pituutta, mutta käyttäjiämme ajatellen sen ei tulisi olla liian pitkä, kuten gosh_this_is_an_impossibly_long_name_to_type.

C++ varaa joukon sanoja kielen käyttöön avainsanoina. Avainsanojen tunnuksia ei saa käyttää uudelleen ohjelman tunnuksina. Olemme jo nähneet lukuisia kielen avainsanoja. Taulukossa 3.1 on lueteltuna kaikki C++:n avainsanat.

Taulukko 3.1 C++:n avainsanat

asm	auto	bool	break	case
catch	char	class	const	const_cast
continue	default	delete	do	double
dynamic_cast	else	enum	explicit	export
extern	false	float	for	friend
goto	if	inline	int	long
mutable	namespace	new	operator	private
protected	public	register	reinterpret_ cast	return
short	signed	sizeof	static	static_cast
struct	switch	template	this	throw
true	try	typedef	typeid	typename
union	unsigned	using	virtual	void
volatile	wchar_t	while		

On olemassa lukuisa joukko yleisesti hyväksyttyjä käytäntöjä olioiden nimeämiseen ja pääosin ne keskittyvät ohjelmien luettavuuteen.

- Olion nimi kirjoitetaan tavallisesti pienillä kirjaimilla. Saatetaan kirjoittaa esimerkiksi index eikä Index tai INDEX. (Tunnuksen Index otaksutaan olevan tyypin nimi ja INDEX:in on yleensä otaksuttu olevan vakioarvo; usein määritelty esikääntäjän #define-direktiivillä.)
- Tunnuksen nimen tulee olla *mnemoninen* — sellainen, että nimi ilmaisee jollain tavalla sen käyttöä ohjelmassa kuten on_lainassa tai palkka. Kuitenkin, jos joku kirjoittaa table tai tbl, on kyse ennemminkin tyylistä kuin oikeellisuudesta.
- Monisanainen tunniste sisältää perinteisesti joko alaviivan sanojen välissä tai ison alkukirjaimen jokaisessa upotetussa sanassa. Esimerkiksi joku voi kirjoittaa opiskelijan_lainaus tai opiskelijanLainaus eikä opiskelijanlainaus (vaikka ainakin Stanin tiedetään kirjoittaneen kaikilla kolmella tavalla). Yleensä ihmiset, joilla on OlioKeskeinenTausta, pitävät isoja alkukirjaimia parempana vaihtoehtona ja ne, joilla on C_tai_proseduraalinen_tausta, suosivat alaviivaa. (Jälleen kerran: siinä, kirjoitetaanko is_a, isA tai isa, on ennemminkin kyse tyylistä kuin oikeellisuudesta.)

3.2.3 Olion määrittely

Yksinkertainen määrittely muodostuu tyyppimääreestä ja nimestä. Määrittely päättyy puolipisteeseen. Esimerkiksi:

```
double salary;
double wage;
int month;
int day;
int year;
unsigned long distance;
```

Silloin, kun yksi tai useampi samaa tyyppiä oleva tunnus määritellään, tyyppimääreen jälkeen voidaan luetella tunnukset pilkuin erotettuna toisistaan. Luettelo voi jakautua useammalle riville. Se päättyy puolipisteeseen. Esimerkiksi edelliset määrittelyt voidaan kirjoittaa uudelleen seuraavasti:

```
double salary, wage;
int month,
   day, year;
unsigned long distance;
```

Yksinkertaisessa määrittelyssä on muuttujan tyyppi ja tunnus. Se ei anna tunnukselle ensimmäistä arvoa eli alkuarvoa. Jos muuttuja on määritelty globaalilla viittausalueella, on taattu, että sille annetaan alkuarvoksi nolla. Esimerkkimme oliot salary, wage, month, day, year ja distance saavat kaikki alkuarvokseen nollan, koska ne on määritelty globaalilla viittausalueella. Jos muuttuja on määritelty paikallisella viittausalueella tai varattu dynaamisesti new-lauseketta käyttäen, ei sille anneta alkuarvoksi nollaa. Näitä olioita kutsutaan *alustamattomiksi*. Alustamaton olio ei ole ilman arvoa, sen sijaan sen arvo on *tuntematon*. (Sanatarkasti ottaen: siihen liittyvä muistialue sisältää satunnaisen bittimallin aikaisemman käytön jäljiltä.)

Koska alustamattoman olion käyttö on yleinen ohjelmointivirhe ja usein vaikea selvittää, suosittelemme yleensä, että jokaiselle määritellylle oliolle annetaan alkuarvo. (Joissakin tapauksissa tämä on tarpeetonta; kuitenkin, ennen kuin kykenet havaitsemaan nuo tapaukset, on alkuarvon antaminen turvallisempaa.) Luokkamekanismissa on automaattinen luokan olioiden alustus, joka tapahtuu *oletusmuodostajan* avulla (tämä esiteltiin kohdassa 2.3). Näemme tämän käsittelyssämme myöhemmin tässä luvussa, kun puhumme vakiokirjaston string-merkkijonon ja kompleksinumeron luokkatyypeistä (kohdat 3.11 ja 3.15). Tässä vaiheessa pane merkille, että seuraavassa

```
int main() {
   // alustamaton paikallinen olio
   int ival;

   // alustettu oletusarvoisessa string-muodostajassa
   string project;

   // ...
}
```

ival on alustamaton paikallinen muuttuja, mutta project on alustettu luokkaolio — alustettu automaattisesti merkkijonoluokan (string) oletusmuodostajassa.

Olion määrittelyssä voidaan sille määrittää alkuarvo. Oliota, jolle on esitelty alkuarvo, kutsutaan *alustetuksi*. C++ tukee kahta muotoa muuttujan alustuksessa. Ensimmäinen muoto on eksplisiittinen syntaksi, jossa käytetään sijoitusoperaattoria:

```
int ival = 1024;
string project = "Fantasia 2000";
```

Implisiittisessä muodossa alkuarvo sijoitetaan sulkujen sisällä:

```
int ival( 1024 );
string project( "Fantasia 2000" );
```

Molemmissa tapauksissa ival alustetaan arvolla 1024 ja project arvolla “Fantasia 2000”.

Pilkuin erotettu tunnusluettelo myös antaa eksplisiittisen alkuarvon jokaiselle oliolle. Syntaksi on seuraava:

```
double salary = 9999.99, wage = salary + 0.01;
int month = 08,
    day = 07, year = 1955;
```

Olion nimi tulee näkyväksi välittömästi tunnuksen määrittelyn jälkeen, ja siten on sallittua, ei kuitenkaan viisasta, alustaa olio itsellään. Esimerkiksi:

```
// sallittua, mutta ei viisasta
int bizarre = bizarre;
```

Lisäksi sisäiset tietotyypit tukevat erityistä *muodostaja*-syntaksia, kun olioita alustetaan nollalla. Esimerkiksi:

```
    // asettaa ival-muuttujaan arvon 0, dval-muuttujaan 0.0
  int ival = int();
double dval = double();
```

Seuraavassa määrittelyssä

```
     // int():iä käytetään jokaiseen kymmeneen elementtiin
vector< int > ivec( 10 );
```

käytetään int()-alustusta automaattisesti jokaiseen kymmeneen ivec:in sisältämään elementtiin. (Kohdassa 2.8 esiteltiin vektorit. Kohdassa 3.10 ja luvussa 6 käsitellään niitä yksityiskohtaisemmin.)

Olio voidaan alustaa millä tahansa monimutkaisella lausekkeella, myös funktion paluuarvolla. Esimerkiksi:

```
#include <cmath>
#include <string>

double price = 109.99, discount = 0.16;
double sale_price( price * discount );
```

```
string pet( "wrinkles" );

extern int get_value();
int val =  get_value();

unsigned abs_val = abs( val );
```

abs(), joka on esimääritelty funktio vakio C:n matematiikkakirjastosta, palauttaa argumenttinsa absoluuttisen arvon. get_value() on käyttäjän määrittelemä funktio, joka palauttaa satunnaisen kokonaislukuarvon.

Harjoitus 3.3

Mitkä seuraavista eivät ole kelvollisia määrittelyjä, vai onko yksikään? Korjaa kaikki ne, jotka havaitsit kelpaamattomiksi.

```
(a) int car = 1024, auto = 2048;
(b) int ival = ival;
(c) int ival( int() );
(d) double salary = wage = 9999.99;
(e) cin >> int input_value;
```

Harjoitus 3.4

Mitkä ovat lvaluen ja rvaluen erot? Anna esimerkki molemmista.

Harjoitus 3.5

Selitä seuraavien kahden ilmentymän, students ja name, väliset erot:

```
(a)  extern string name;
     string name( "harjoitus 3.5a" );

(b)  extern vector<string> students;
     vector<string> students;
```

Harjoitus 3.6

Mitkä seuraavista nimistä eivät ole kelvollisia, vai onko yksikään? Korjaa jokainen huomaamasi kelpaamaton nimi.

```
(a) int double = 3.14159; (b) vector< int > _;
(c) string namespace;     (d) string catch-22;
(e) char 1_or_2 = '1';    (f) float Float = 3.14f;
```

Harjoitus 3.7

Mitkä ovat erot seuraavan globaalin ja paikallisen olion määrittelyssä, jos niitä on?

```
string global_class;
int global_int;
```

```
int main() {
   int local_int;
   string local_class;

   // ...
}
```

3.3 Osoitintyypit

Esittelimme lyhyesti osoittimet ja dynaamisen muistinvarauksen kohdassa 2.2. Osoitin pitää sisällään toisen olion osoitteen, joka mahdollistaa tuon olion epäsuoran käsittelyn. Tyypillisiä osoittimen käyttökohteita ovat linkitettyjen tietorakenteiden kuten puiden ja listojen luominen, dynaamisesti varattujen olioiden hallinta ohjelman suorituksen aikana ja funktion parametrityyppinä pääasiassa taulukoiden ja suurten luokkien olioiden välittäminen.

Jokaiseen osoittimeen liittyy tyyppi. Erilaisten tietotyyppien osoittimien väliset erot eivät ole osoittimen esitystavassa eivätkä arvoissa (osoitteissa), jotka ne pitävät sisällään — nämä ovat yleensä samoja kaikille tiedon osoittimille[1]. Ero on sen sijaan olion tyypissä, jota osoitetaan. Osoittimen tyyppi ohjailee sitä, kuinka kääntäjää tulkitsee muistia, joka löytyy tietystä osoitteesta, kuten myös sitä, kuinka paljon muistia tuo tulkinnan tulos vie.

- int-osoitin, joka osoittaa muistipaikkaan 1000 32-bittisessä koneessa, vie muistialueen 1000 – 1003.
- double-osoitin, joka osoittaa muistipaikkaan 1000 32-bittisessä koneessa, vie muistialueen 1000 – 1007.

Tässä on joitakin esimerkkejä osoittimien määrittelyistä:

```
int           *ip1, *ip2;
complex<double> *cp;
string        *pstring;
vector<int>    *pvec;
double        *dp;
```

Osoitin määritellään laittamalla tunnuksen eteen käänteisoperaattori (*). Pilkuin erotellussa määrittelyluettelossa pitää käänteisoperaattori olla jokaisen sellaisen tunnuksen edessä, jonka haluaisi toimia osoittimena. Seuraavassa esimerkissä lp tulkitaan osoittimeksi long-tyyppiseen olioon ja lp2 tulkitaan long-tyyppiseksi tieto-olioksi eikä osoittimeksi:

```
long *lp, lp2;
```

Tässä esimerkissä fp tulkitaan float-tyyppiseksi tieto-olioksi ja fp2 tulkitaan osoittimeksi float-tyyppiin:

```
float fp, *fp2;
```

1. Tämä ei varsinaisesti päde funktio-osoittimiin, jotka osoittavat ohjelman koodisegmenttiin. Funktio-osoittimet ja tieto-osoittimet ovat erilaisia. Osoittimia funktioihin käsitellään kohdassa 7.9.

Selvyyden vuoksi on parempi kirjoittaa

```
string *ps;
```

seuraavan sijasta

```
string* ps;
```

Mahdollisuus on, että ohjelmoija, joka myöhemmin haluaa määritellä toisen merkkijono-osoittimen, voi virheellisesti muokata tätä määrittelyä seuraavasti:

```
// hups: ps2 ei ole merkkijono-osoitin
string* ps, ps2;
```

Osoitin voi sisältää arvon 0, joka ilmaisee, että se ei osoita mihinkään olioon eli samantyyppiseen tieto-olion osoitteeseen. Jos on annettu määrittely ival-oliolle

```
int ival = 1024;
```

ovat seuraavat kahden osoittimen — pi ja pi2 — määrittelyt ja sijoitukset kelvottomia:

```
// pi alustetaan osoittamaan ei mihinkään olioon
int *pi = 0;

// pi2 alustetaan osoittamaan ival-oliota
int *pi2 = &ival;

// ok: pi ja pi2 molemmat osoittavat nyt ival-olioon
pi = pi2;

// pi2 ei osoita nyt mitään oliota
pi2 = 0;
```

Osoitin ei voi sisältää osoitteetonta arvoa. Esimerkiksi seuraava sijoitus johtaa käännöksenaikaiseen virheeseen:

```
// virhe: pi:hin sijoitettu ival-olion int-arvo
pi = ival;
```

Eikä osoitinta voi myöskään alustaa tai siihen sijoittaa toisentyyppisen olion osoitetta. Esimerkiksi molemmat seuraavat määrittelyt

```
double dval;
double *pd = &dval;
```

johtavat käännöksenaikaiseen virheeseen:

```
// molemmat ovat käännöksenaikaisia virheitä
// kelvoton tyyppisijoitus: int* <== double*
pi = pd;
pi = &dval;
```

Eikö pi voisi fyysisesti sisältää muistiosoitteen, joka liittyy dval-olioon? Voi se. Mutta se ei ole sallittua, koska vaikka pi ja pd pystyvät sisältämään saman osoitteen arvon, ne tulkitsevat tuon muistin sommitelman ja laajuuden eri tavalla.

Tietysti jos kaikki, mitä haluamme tehdä, on pitää osoitteen arvo (ja mahdollisesti verrata osoitteen arvoa toiseen osoitearvoon), silloin osoittimen varsinainen tyyppi ei haittaa. Juuri tuohon on olemassa erityinen osoitintyyppi: void*-osoittimeen voidaan sijoittaa minkä tahansa tietotyypin osoitteen arvo (funktion osoitinta siihen ei pysty sijoittamaan).

```
// ok: void* voi sisältää
// minkä tahansa osoitintyypin osoitteen arvon
void *pv = pi;
pv = pd;
```

void* ilmaisee, että siihen liittyvä arvo on osoite, mutta että tuossa osoitteessa olevan olion tyyppi on tuntematon. Me emme voi operoida oliota, jota void*-osoitin osoittaa. Voimme vain siirtää tuon osoitteen arvon tai verrata sitä toiseen osoitteen arvoon. (Katsomme void*-osoitinta tarkemmin kohdassa 4.14.)

Kun on annettu int-osoitinolio pi, niin kirjoittamalla pi

```
// saa arvokseen pi:n sisältämän osoitteen
// tyyppi: int*
pi;
```

tarkoitetaan osoitteen arvoa, jonka pi juuri nyt sisältää. Kirjoittamalla &pi

```
// saa arvokseen pi:n sisältämän todellisen osoitteen
// tyyppi: int**
&pi;
```

tarkoitetaan osoitetta, johon osoitinolio pi on tallennettu. Kuinka käsittelemme oliota, jota pi osoittaa?

Oletusarvo on, että ei ole olemassa tapaa päästä varsinaiseen olioon, jota pi osoittaa, eikä lukea tai kirjoittaa olioon. Jotta voisimme käsitellä osoittimen osoittamaa oliota, meidän pitää käyttää osoitinta käänteisesti (*dereference*). On olemassa erityinen *käänteisoperaattori* (*), jolla päästään epäsuorasti lukemaan ja kirjoittamaan osoitettuun olioon. Olkoon esimerkiksi seuraavat määrittelyt:

```
int ival = 1024, ival2 = 2048;
int *pi = &ival;
```

Seuraavassa on esimerkki, kuinka voisimme käyttää pi:tä käänteisesti, jotta pääsisimme käsittelemään ival-oliota epäsuorasti:

```
// käytä pi:tä käänteisesti ja sijoita olio, jota se osoittaa, joka tässä tapauksessa on
// ival, ival2:n rvalue
*pi = ival2;

// oikeanpuoleinen ilmentymä lukee pi:n osoittaman olion
// arvon; vasemmanpuoleinen ilmentymä
// kirjoittaa oikeanpuoleisen lausekkeen olioon

*pi = abs( *pi ); // ival = abs(ival);
*pi = *pi + 1;    // ival = ival + 1;
```

Tiedämme, että kun otamme int-tyyppisen olion osoitteen

```
int *pi = &ival;
```

on tuloksena int* — tarkoittaa osoitinta int-tyyppiin. Kun otamme osoitteen tyyppiosoittimesta int-tyyppiin:

```
int **ppi = &pi;
```

on tuloksena int** — tarkoittaa osoitinta int-tyypin osoittimeen. Kun käytämme ppi:tä käänteisesti

```
int *pi2 = *ppi;
```

saamme osoitteen arvon, jonka ppi-osoitin sisältää — tässä tapauksessa pi:n sisältämä arvo, joka on ival:in osoite. Jotta todella pääsisimme käsittelemään ival:ia, pitää ppi:tä käyttää käänteisesti kahdesti. Esimerkiksi:

```
cout << "ival:in arvo\n"
    << "suora arvo: " << ival << "\n"
    << "epäsuora arvo: " << *pi << "\n"
    << "kahdesti epäsuora arvo: " << **ppi
    << endl;
```

Seuraavat kaksi sijoituslausetta käyttäytyvät melkoisesti eri tavalla, vaikka ne molemmat ovat sallittuja. Ensimmäinen lause kasvattaa tieto-olion arvoa, jota pi osoittaa; toinen kasvattaa osoitteen arvoa, jonka osoitin pi sisältää.

```
int i, j, k;
int *pi = &i;

// lisää kaksi i:hin (i = i + 2)
*pi = *pi + 2;

// lisää osoitteeseen, jonka pi sisältää
pi = pi + 2;
```

Osoitteen arvoa voidaan lisätä tai vähentää kokonaisarvolla. Tämänkaltainen osoittimen käsittely, jota sanotaan *osoitinaritmetiikaksi*, voi aluksi vaikuttaa hieman vaikealta käsittää, kunnes huomaamme, että lisäys koskee tieto-olioita eikä irrallisia desimaaliarvoja. Arvon 2 lisääminen osoittimeen kasvattaa sen osoittaman osoitteen arvoa, jonka koko on kaksi sen tyyppistä oliota. Olettaen esimerkiksi, että char on 1 tavu, int on 4 tavua ja double on 8 tavua, arvon 2 lisääminen osoittimeen kasvattaa sen osoitetta arvolla 2, 8 tai 16 riippuen siitä, onko osoittimen tyyppi char, int vai double.

Käytännössä osoitearitmetiikan taataan toimivan hyvin vain, jos osoitin osoittaa taulukon elementtiä. Edellisessä esimerkissä ei ole taattua, että kolme kokonaislukumuuttujaa ovat tallennettuina yhtenäisesti muistiin; sen vuoksi ip+2 saattaa tai ei saata johtaa kelvolliseen osoitteeseen riippuen siitä, mitä todellisuudessa on tallennettu tuohon paikkaan. Osoitinaritmetiikan tyypillinen käyttötapa on taulukon läpikäynti. Esimerkiksi:

```
int ia[ 10 ];
```

```
int *iter = &ia[0];
int *iter_end = &ia[10];

while ( iter != iter_end ) {
   do_something_with_value( *iter );
   ++iter; // nyt iter osoittaa seuraavaan jäseneen
}
```

Harjoitus 3.8

Kun on annettu seuraavat määrittelyt

```
int ival = 1024, ival2 = 2048;
int *pi1 = &ival, *pi2 = &ival2, **pi3 = 0;
```

selitä, mitä tapahtuu seuraavissa sijoituksissa. Yksilöi ne, jotka ovat virheellisiä, jos sellaisia on.

```
(a) ival = *pi3;  (e) pi1 = *pi3;
(b) *pi2 = *pi3;  (f) ival = *pi1;
(c) ival = pi2;   (g) pi1 = ival;
(d) pi2 = *pi1;   (h) pi3 = &pi2;
```

Harjoitus 3.9

Osoittimet ovat C- ja C++-ohjelmoinnin tärkeä piirre ja ovat siten yleisiä ohjelmavirheiden lähteitä. Esimerkiksi

```
pi = &ival2;
pi = pi + 1024;
```

melkein takaa, että se jättää `pi`:n osoittamaan satunnaiselle muistialueelle. Mitä sijoitus tekee ja milloin se ei olisi virhe?

Harjoitus 3.10

Samalla tavalla seuraavan pienen ohjelman käyttäytyminen on tuntematon ja todennäköisesti menee virheeseen suorituksen aikana:

```
int foobar( int *pi ) {
   *pi = 1024;
   return *pi;
}

int main()
{
```

```
    int *pi2 = 0;
    int ival = foobar( pi2 );
    return 0;
}
```

Mikä tässä aiheuttaa ongelman? Miten korjaisit sen?

Harjoitus 3.11

Edellisissä kahdessa esimerkissä tapahtuu virheitä, koska osoittimen käytöstä puuttuu suoritusaikainen tarkistus. Jos osoittimilla on niin merkittävä osa C++-ohjelmoinnista, minkä ajattelet olevan syynä siihen, että osoittimien käyttöön ei ole rakennettu enempää turvallisuutta? Voitko ajatella yhtään yleistä ohjetta, jolla osoittimien käyttöä saisi turvallisemmaksi?

3.4 Merkkijonotyypit

C++:ssa on kaksi merkkijonon esitystapaa: C-tyylinen merkkijono ja string-merkkijonoluokkatyyppi, joka on esitelty C++-standardissa. Yleensä suosittelemme merkkijonoluokan käyttöä, mutta käytännössä on yhä monia ohjelmatilanteita, joissa on joko hyödyllistä tai välttämätöntä ymmärtää ja käyttää vanhempaa C-tyylistä merkkijonoa. Eräs esimerkki, jonka näemme luvussa 7, on komentorivin valitsimien välitys `main()`-funktiolle C-tyylisellä merkkijonolla.

3.4.1 C-tyylinen merkkijono

C-tyylinen merkkijono on peräisin C-kielestä ja sen tukea jatketaan C++:ssa. (Itse asiassa ennen C++-standardia se oli ainoa käytettävissä oleva merkkijono pois luettuna kolmannen osapuolen kirjastoluokat.)

Merkkijono on tallennettuna merkkitaulukkoon ja sitä yleensä käsitellään `char*`-osoittimen kautta. C:n vakiokirjastossa on C-tyylisten merkkijonojen käsittelyyn joukko funktioita kuten seuraavat:

```
  // palauttaa merkkijonon pituuden
int strlen( const char* );

// vertaa kahden merkkijonon yhtäsuuruutta
int strcmp( const char*, const char* );

// kopioi toisen merkkijonon ensimmäiseen
 char* strcpy(char*, const char* );
```

(C:n vakiokirjasto on otettu mukaan osaksi C++-standardia.) Jotta näitä funktioita voitaisiin käyttää, pitää ottaa mukaan siihen liittyvä C-otsikkotiedosto:

```
#include <cstring>
```

Merkkiosoitin, joka osoittaa C-tyyliseen merkkijonoon, viittaa aina siihen liittyvään merkkitaulukkoon. Vaikka kirjoittaisimme merkkijonoliteraalin kuten

```
const char *st = "The expense of spirit\n";
```

järjestelmä tallentaa merkkijonon sisäisesti merkkitaulukkoon. Sitten st osoittaa tuon taulukon ensimmäiseen elementtiin. Kuinka voisimme käsitellä st:tä merkkijonona?

Yleensä käymme C-tyylisen merkkijonon läpi käyttäen osoitinaritmetiikkaa ja lisäämme osoittimeen arvoon 1 niin monta kertaa, että saavutamme päättävän null-merkin. Esimerkiksi:

```
while ( *st++ ) { ... }
```

char*-osoitinta st käytetään käänteisesti ja osoitettua merkkiä testataan true- tai false-arvoon (tosi/epätosi). true-arvo on mikä tahansa merkki, joka on muu kuin null-merkki. Ilmaisu ++ on lisäysoperaattori ja se kasvattaa st:tä osoittamaan seuraavan taulukon merkkiin.

Osoitinta käytettäessä on yleensä välttämätöntä testata, että se osoittaa johonkin olioon ennen kuin sitä käytetään käänteisesti. Muutoin ohjelma tekee todennäköisesti virheen. Esimerkiksi:

```
int
string_length( const char *st )
{
        int cnt = 0;
        if ( st )
            while ( *st++ )
                  ++cnt;
        return cnt;
}
```

C-tyylisen merkkijonon pituus voi olla nolla (ja sitä kohdellaan siten tyhjänä) kahdella eri tavalla: merkkiosoitin voi olla asetettu nollaksi eikä siksi osoita mihinkään olioon; vaihtoehtoisesti osoitin voi olla asetettu, mutta merkkijono, johon se viittaa, sisältää vain null-merkin. Esimerkiksi:

```
// pc1 ei osoita mihinkään olioon
char *pc1 = 0;

// pc2 osoittaa null-merkkiä
const char *pc2 = "";
```

C-tyyliset merkkijonot yleensä osoittautuvat virhealttiiksi aloitteleville C- tai C++-ohjelmoijille, mikä johtuu niiden matalan tason esitystavasta. Seuraavissa koodikatkelmissa käymme läpi monia vasta-alkajien tyypillisiä virheitä. Ohjelman tehtävä on selkeä: laske st:n pituus. Valitettavasti ensimmäinen yrityksemme on virheellinen. Huomaatko virheen?

```
#include <iostream>
const char *st = "The expense of spirit\n";
```

```
int main() {
   int len = 0;
   while ( st++ ) ++len;

   cout << len << ": " << st;
   return 0;
}
```

Ohjelma tekee virheen, koska st:tä ei käytetä käänteisesti — tämä tarkoittaa, että

```
st++
```

testaa, onko st:n osoite 0 vai ei eikä sitä, onko sen osoittaman merkin arvo null. Ehto on aina tosi, koska silmukka lisää jokaisella toistokerralla st:n osoitteeseen arvon 1. Ohjelma jatkaa suoritustaan ikuisesti tai kunnes järjestelmä pysäyttää sen. Tällaisen silmukan sanotaan olevan *päättymätön* (*ikuinen*) *silmukka*.

Toinen versiomme ohjelmasta korjaa tämän erehdyksen. Ohjelma suoritetaan loppuun saakka. Valitettavasti sen tulostus on virheellinen. Huomaatko, minkä virheen olemme tehneet tällä kerralla?

```
#include <iostream>
const char *st = "The expense of spirit\n";

int main()
{
   int len = 0;
   while ( *st++ ) ++len;

   cout << len << ": " << st << endl;
   return 0;
}
```

Virhe on tässä tapauksessa se, että st ei enää osoita merkkijonoliteraalivakioon. Sitä on kasvatettu yksi merkki yli sen päättävän null-merkin. (Ohjelman tulostus riippuu osoitetun muistin sisällöstä.) Tässä on yksi mahdollinen ratkaisu:

```
st = st - len;
cout << len << ": " << st;
```

Ohjelma käännetään ja suoritetaan. Tulostus on kuitenkin yhä virheellinen. Nyt se generoi seuraavaa:

```
22: he expense of spirit
```

Tämä heijastaa jotain ohjelmoinnin luonteesta. Huomaatko, minkä virheen olemme tehneet tällä kerralla?

Merkkijonon päättävää null-merkkiä ei ole otettu huomioon, kun on laskettu merkkijonon pituutta. st:n osoitetta pitää positioida uudelleen merkkijonon pituudella *plus 1*. Seuraava rivi on oikein:

```
st = st - len - 1;
```

Kun ohjelma käännetään ja suoritetaan, ohjelma generoi seuraavan, joka on lopultakin oikea tulostus:

```
22: The expense of spirit
```

Ohjelma on nyt oikein. Ohjelmoinnin tyylin termein se on kuitenkin yhä hieman epäelegantti. Lause

```
st = st - len - 1;
```

on lisätty korjaamaan virhettä, jonka sai aikaan st:n suora lisääminen. Uudelleensijoitus st:hen ei kuitenkaan sovi ohjelman alkuperäiseen logiikkaan ja ohjelmaa on hieman vaikeampi ymmärtää.

Ohjelman korjausta kuten tässä sanotaan usein *paikkaukseksi* (*patch*) — levitetään jotain "reiän" päälle olemassaolevaan ohjelmaan. Paikkasimme ohjelmaamme hyvittääksemme logiikkavirheen alkuperäisessä suunnitelmassamme. Parempi ratkaisu on korjata alkuperäinen suunnitteluvirhe. Yksi ratkaisu on määritellä toinen merkkiosoitin ja alustaa se st:llä. Esimerkiksi:

```
const char *p = st;
```

p:tä voidaan nyt käyttää st:n pituuden laskemiseen ja itse st jää muuttumattomaksi:

```
while ( *p++ )
```

3.4.2 String-tyyppi

Kuten olemme nähneet, merkkiosoittimen käyttö merkkijonon esittämiseen on helppo tapa tehdä virheitä, koska sen on matalan tason esitystapa. Jotta ohjelmoijia olisi suojeltu monilta yleisiltä sudenkuopilta, joita C-tyylisiin merkkijonoihin liittyy, ei ollut epätavallista, että jokaisella projektilla, osastolla tai yrityksellä oli oma merkkijonoluokkansa — itse asiassa tämän tekstin kaksi ensimmäistä painosta teki juuri niin! Ongelma on, että jokainen tekee oman merkkijonototeutuksensa ja ohjelmien siirrettävyydestä ja yhteensopivuudesta tulee huomattavasti vaikeampaa. C++:n vakiokirjastossa on yleinen merkkijonon toteutus luokka-abstraktiona.

Millaisia operaatioita voisit odottaa merkkijonoluokalta? Mistä muodostuu vähimmäisjoukko sen olennaisesta käyttäytymisestä?

1. Tuki merkkijono-olion alustamiseen sekä peräkkäisillä merkeillä ja toisella merkkijono-oliolla. C-tyylisessä merkkijonossa yhden merkkijonon alustamista toisella ei ole tuettu.
2. Tuki kopioida yksi merkkijono toiseen. C-tyylisessä merkkijonossa tämä saavutetaan käyttämällä kirjastofunktiota strcpy().
3. Tuki yksittäisten merkkien lukemiseen ja kirjoittamiseen. C-tyylisessä merkkijonossa yksittäisten merkkien käsittely tapahtuu indeksioperaattorin kautta tai käyttämällä suoraa osoitinta käänteisesti.
4. Tuki verrata kahden merkkijonon yhtäsuuruutta. C-tyylisessä merkkijonossa vastaava vertailu tapahtuu kirjastofunktion strcmp() avulla.

5. Tuki kahden merkkijonon liittämiseksi yhteen joko liittämällä yksi merkkijono toiseen tai yhdistämällä kaksi merkkijonoa kolmannen muotoiseksi. C-tyylisessä merkkijonossa yhdistely tapahtuu kirjastofunktion strcat() kautta. Kahden merkkijonon yhdistäminen kolmanneksi toteutetaan kopioimalla yksi merkkijono uudeksi ilmentymäksi käyttämällä strcpy()-funktiota ja sitten yhdistämällä toinen merkkijono uuteen ilmentymään käyttämällä strcat()-funktiota.
6. Tuki tiedolle, kuinka monta merkkiä merkkijono sisältää. C-tyylisessä merkkijonossa sen pituus palautetaan kirjastofunktiolla strlen().
7. Tuki tiedolle, onko merkkijono tyhjä vai ei. C-tyylisessä merkkijonossa tämä toteutetaan seuraavalla kaksivaiheisella testillä:

```
char *str = 0;
//...
if ( ! str || ! *str )
      return;
```

C++:n vakiokirjastossa on string-merkkijonoluokkatyyppi, joka tukee näitä operaatioita (ja paljon muuta, kuten näemme luvussa 6). Tässä vaiheessa katsomme, kuinka merkkijonotyyppi tukee näitä operaatioita.

Jotta merkkijonotyyppiä voidaan käyttää, pitää ottaa mukaan siihen liittyvä otsikkotiedosto:

```
#include <string>
```

Esimerkiksi tässä on aikaisempi merkkijonomme määritelty uudelleen string-tyyppiseksi:

```
#include <string>
string st( "The expense of spirit\n" );
```

st:n pituus palautetaan sen size()-operaatiolla (siihen ei kuulu mukaan päättävää null-merkkiä):

```
cout << "Merkkijonon "
   << st
   << " koko on " << st.size()
   << " merkkiä, mukaan lukien rivinvaihtomerkki\n";
```

Toinen muoto merkkijonon muodostuksesta määrittelee tyhjän merkkijonon. Esimerkiksi:

```
string st2; // tyhjä merkkijono
```

Kuinka voimme olla varmoja sen tyhjyydestä? Yksi tapa on tietysti testata sen kokoa arvoon 0:

```
if ( ! st.size() )
   // ok: tyhjä
```

Suorempi metodi on empty()-operaatio:

```
if ( st.empty() )
   // ok: tyhjä
```

empty() palauttaa bool-vakion true, jos merkkijono ei sisällä merkkejä; muussa tapauksessa se palauttaa arvon false.

Kolmas merkkijonon muodostamistapa alustaa yhden merkkijono-olion toisella. Esimerkiksi

```
string st3( st );
```

alustaa st3:n niin, että siitä tulee st:n kopio. Kuinka voimme tarkistaa sen? Yhtäsuuruusoperaattori vertaa kahden merkkijonon yhtäsuuruutta ja palauttaa arvon true, jos ne ovat yhtäsuuria:

```
if ( st == st3 )
    // alustus toimi
```

Kuinka voimme kopioida merkkijonon toiseen? Yksinkertaisin metodi on käyttää sijoitusoperaattoria. Esimerkiksi seuraavassa

```
st2 = st3; // kopioi st3 st2:een
```

poistetaan muistialue, joka sisältää st2:een liittyvät merkit, varataan muisti, joka tarvitaan merkeille, jotka liittyvät st3:een ja sitten kopioidaan st3:n merkit.

On myös mahdollista yhdistää kaksi tai useampia merkkijonoja käyttäen joko plus-operaattoria (+) tai hieman oudolta näyttävää yhdistettyä sijoitusoperaattoria (+=). Jos esimerkiksi on annettu kaksi merkkijonoa

```
string s1( "hei, " );
string s2( "maailma\n" );
```

voimme yhdistää nuo kaksi merkkijonoa luomalla niistä kolmannen merkkijonon kuten seuraavassa:

```
string s3 = s1 + s2;
```

Jos sen sijaan haluaisimme lisätä s2:n suoraan s1:een, silloin käyttäisimme +=-operaattoria:

```
s1 += s2;
```

s1:n ja s2:n alustuksemme on tehty hieman kömpelösti, kun sisällytimme niihin tyhjän merkin, pilkun ja rivinvaihtomerkin. Niiden läsnäolo rajoittaa merkkijono-olioiden uudelleenkäytettävyyttä, vaikka ne toimivat välittömään tarpeeseemme. Vaihtoehtona on yhdistellä string-olioita ja C-tyylisiä merkkijonoja kuten seuraavassa:

```
const char *pc = ", ";
string s1( "hei" );
string s2( "maailma" );

string s3 = s1 + pc + s2 + "\n";
```

Tällainen yhdistely on parempi, koska se jättää s1:n ja s2:n yleisemmin uudelleenkäytettävään muotoon. Se toimii, koska string-tyyppi pystyy automaattisesti konvertoimaan C-tyylisen merkkijonon string-olioksi. Tämä esimerkiksi mahdollistaa C-tyylisen merkkijonon sijoittamisen string-olioon:

```
string s1;
const char *pc = "merkkitaulukko";

s1 = pc; // ok
```

Päinvastaista konversiota ei kuitenkaan tehdä automaattisesti. Ei ole olemassa tukea string-olion implisiittiselle konvertoimiselle C-tyyliseksi merkkijonoksi. Esimerkiksi seuraava yritys alustaa `str` string-oliolla `s1` saa aikaan käännöksenaikaisen virheen:

```
char *str = s1; // käännöksenaikainen tyyppivirhe
```

Jotta tämä konversio saadaan aikaan, pitää käynnistää eksplisiittisesti oudosti nimetty `c_str()`-operaatio:

```
char *str = s1.c_str(); // melkein ok, mutta ei ihan
```

Nimi `c_str()` on viittaus suhteeseen string-tyypistä C-tyylisen merkkijonon esitystapaan. Kirjaimellisesti ottaen se tarkoittaa: “anna minulle C-tyylisen merkkijonon esitystapa” — eli merkkiosoitin merkkitaulukon alkuun.

Alustus kuitenkin yhä epäonnistuu, mutta nyt eri syystä: `c_str()` palauttaa osoittimen vakiotaulukkoon estääkseen taulukon käsittelyn epäsuorasti ohjelmasta (seuraavassa kohdassa tutkitaan `const`-määrettä):

```
const char*
```

`str` on määritelty muuksi kuin vakio-osoittimeksi, joten sijoitus torjutaan tyyppivirheenä. Oikea alustus on seuraava:

```
const char *str = s1.c_str(); //ok
```

Merkkijonotyyppi tukee yksittäisten merkkien käsittelyä indeksioperaattorin kautta. Esimerkiksi seuraavassa koodikatkelmassa kaikki merkkijonossa olevat pisteet korvataan alaviivalla:

```
string str( "fa.disney.com" );

int size = str.size();
for ( int ix = 0; ix < size; ++ix )
    if ( str[ ix ] == '.' )
        str[ ix ] = '_';
```

Tähän päättyy kaikki, mitä haluamme sanoa merkkijonoista tässä vaiheessa, vaikka paljon on vielä sanomatta. Esimerkiksi seuraavassa on vaihtoehtoinen toteutus edellä olevalle koodikatkelmalle:

```
replace( str.begin(), str.end(), '.', '_' );
```

`replace()` on yksi geneerisistä algoritmeista, jotka esittelimme lyhyesti kohdassa 2.8 (ja joita katsomme yksityiskohtaisesti luvussa 12 — kirjan liitteessä on aakkosellinen luettelo kaikista algoritmeista yhdessä käyttöesimerkkiensä kanssa).

`begin()`- ja `end()`-operaatiot palauttavat iteraattorit merkkijonon alkuun ja loppuun. Iteraattori on vakiokirjaston luokka-abstraktio osoittimesta (katsoimme iteraattoreita lyhyesti kohdassa

2.8 ja katsomme niitä lisää yksityiskohtaisesti sekä luvussa 6 että luvussa 12).

replace() iteroi begin():in ja end():in välissä olevat merkit. Jokainen merkki, joka on yhtä kuin piste, korvataan alaviivalla.

Harjoitus 3.12

Mitkä seuraavista ovat virheellisiä, vai onko yhtään?

```
(a) char ch = "The long, winding road";
(b) int ival = &ch;
(c) char *pc = &ival;
(d) string st( &ch );

(e) pc = 0;     (i) pc = '0';
(f) st = pc;    (j) st = &ival;
(g) ch = pc[0]; (k) ch = *pc;
(h) pc = st;    (l) *pc = ival;
```

Harjoitus 3.13

Selitä seuraavien kahden while-silmukan välinen ero:

```
while ( st++ )
      ++cnt;

while ( *st++ )
      ++cnt;
```

Harjoitus 3.14

Tarkastelepa seuraavia kahta semanttisesti yhdenmukaista ohjelmaa, joista toinen käyttää C-tyylisiä merkkijonoja ja toinen käyttää string-merkkijonotyyppiä.

```
// ***** C-tyylinen merkkijonon toteutus *****

#include <iostream>
#include <cstring>

int main()
{
        int errors = 0;
        const char *pc = "erittäin pitkä merkkijonoliteraali";

        for ( int ix = 0; ix < 1000000; ++ix )
        {
                int len = strlen( pc );
                char *pc2 = new char[ len + 1 ];
                strcpy( pc2, pc );

                if ( strcmp( pc2, pc ))
```

```
                ++errors;

            delete [] pc2;
        }
        cout << "C-tyyliset merkkijonot: "
            << errors << " virhettä tapahtunut.\n";
}

// ***** string-toteutus *****

#include <iostream>
#include <string>

int main()
{
        int errors = 0;
        string str( "erittäin pitkä merkkijonoliteraali" );

        for ( int ix = 0; ix < 1000000; ++ix )
        {
            int len = str.size();
            string str2 = str;
            if ( str != str2 )
                ++errors;
        }
        cout << "string-luokka: "
            << errors << " virhettä tapahtunut.\n";
}
```

(a) Selitä, mitä ohjelmat tekevät.
(b) Kun nämä ajetaan, on string-luokan toteutuksen suoritus keskimäärin kaksi kertaa niin nopea kuin C-tyylisen merkkijonoluokan. Suhteelliset, keskimääräiset suoritusajat UNIX:issa `timex`-komennolla ovat seuraavat:

```
user      0.96   # string-luokkka
user      1.98   # C-tyylinen merkkijono
```

Oletitko näin? Kuinka selittäisit sen?

Harjoitus 3.15

C++:n merkkijonotyyppi on esimerkki oliopohjaisesta luokka-abstraktiosta. Onko jotain, mitä haluaisit muuttaa sen käytössä tai operaatiojoukossa, jotka on esitetty tässä yhteydessä? Onko olemassa yhtään lisäoperaatiota, jonka uskot olevan välttämätön? Hyödyllinen? Selitä.

3.5 Const-määre

Seuraavassa for-silmukassa on kaksi ongelmaa, jotka molemmat koskevat arvon 512 käyttöä ylärajana.

```
for ( int index = 0; index < 512; ++index )
    ... ;
```

Ensimmäinen ongelma on luettavuus. Mitä tarkoittaa testata index:iä arvoon 512? Mitä silmukka tekee — mikä saa arvon 512 merkitsemään jotain? (Tässä esimerkissä arvon 512 sanotaan olevan *taikanumero* (*magic number*), jonka merkitys ei ole ilmeinen sen käytön yhteydessä. On kuin numero olisi tullut tyhjästä.)

Toinen ongelma on ylläpidettävyys. Kuvitellaan, että ohjelmassa on 10 000 riviä. Arvon 512 käyttöä esiintyy koodissa 4 %. 400 ilmentymästä pitää 80 % konvertoida arvoon 1024. Jotta voisimme sen tehdä, meidän pitää tietää, mitkä pitää konvertoida ja mikä ei. Vaikkapa vain yhden ilmentymän virheellinen muutos rikkoo ohjelman ja edellyttää palaamista takaisin konvertoimiseen ja tutkimaan uudelleen jokaisen arvon käytön.

Ratkaisu molempiin ongelmiin on käyttää oliota, joka on alustettu arvoon 512. Valitsemalla mnemonisen nimen, ehkäpä bufSize, voimme tehdä ohjelmasta luettavamman. Testi tapahtuu nyt oliota vastaan literaalivakion sijasta:

```
index < bufSize
```

320 ilmentymää ei enää tarvitse etsiä ja konvertoida siinä tapauksessa, että bufSize muuttuu. Sen sijaan vain se rivi, jossa bufSize alustetaan, pitää muuttaa. Tämä lähestymistapa ei ainoastaan vähennä merkittävästi työtä, vaan myös virheiden tekemisen mahdollisuus vähenee huomattavasti. Ratkaisun hinta on yksi lisämuuttuja. Arvon 512 sanotaan nyt *paikallistetun*.

```
int bufSize = 512; // syöttöpuskurin koko
// ...

for ( int index = 0; index < bufSize; ++index )
    // ...
```

Tämän ratkaisun ongelma on, että bufSize on lvalue. On mahdollista, että ohjelmassa vahingossa muutetaan sitä. Tässä on esimerkiksi yleinen ohjelmoijan virhe:

```
// muuttaa vahingossa bufSize:n arvoa
if ( bufSize = 1 )
    // ...
```

C++:ssa "=" on sijoitusoperaattori ja "==" on yhtäsuuruusoperaattori. Ohjelmoija on vahingossa muuttanut bufSize:n arvoksi 1 ja tuo johtaa vaikeasti jäljitettävään ohjelmavirheeseen. (Sellaista virhettä on vaikea etsiä, koska ohjelmoija ei *näe* koodin olevan väärin ja tästä syystä monet kääntäjät antavat varoituksen tämänkaltaisista sijoituslausekkeista.)

const-tyyppimääreessä on ratkaisu. Se muuntaa olion *vakioksi* (*constant*). Esimerkiksi

```
const int bufSize = 512; // syöttöpuskurin koko
```

määrittelee bufSize:n vakioksi ja alustaa sen arvolla 512. Jokainen yritys muuttaa tuota arvoa ohjelmassa johtaa käännöksenaikaiseen virheeseen. Tästä syystä sitä kutsutaan *vain-luku*-tyyppiseksi (*read-only*). Esimerkiksi:

```
// virhe: yritys kirjoittaa vakio-olioon
if ( bufsize = 0 ) ...
```

Koska vakiota ei voi muokata määrittelyn jälkeen, pitää pitää xalustaa. Alustamattoman vakion määrittely johtaa käännöksenaikaiseen virheeseen.

```
const double pi; // virhe: alustamaton vakio
```

Kun vakio on määritelty, emme voi muuttaa const-olioon liittyvää arvoa. Voimmeko toisaalta sijoittaa sen osoitteen osoittimeen? Pitäisikö esimerkiksi seuraavan olla sallittua?

```
const double minWage = 9.60;

// ok? virhe?
double *ptr = &minWage;
```

Pitäisikö tämän olla sallittua? minWage on const-olio, joten on taattu, että sen arvoa ei voida kirjoittaa yli uudella arvolla. ptr on kuitenkin tavallinen osoitin eikä mikään estä meitä kirjoittamasta seuraavaksi

```
*ptr += 1.40; // muokattu minWage:a!
```

Kääntäjä ei yleensä pysty pitämään kirjaa, mitä oliota osoitin osoittaa missäkin vaiheessa ohjelman aikana (sellainen seuranta vaatii *tiedonkulun analyysiä*, jota tavallisesti hoitaa erillinen *optimoijakomponentti* (*optimizer*)). Kääntäjän ei ole mahdollista sallia muuttujaksi tarkoitetun osoittimen osoittaa vakio-olioon ja sen tulee ilmoittaa virheenä, jos vain yritetäänkin muuttaa vakio-oliota epäsuorasti sen kautta. Sen sijaan jokainen yritys saada muuttujaksi tarkoitetun osoittimen arvoksi vakio-olion osoite johtaa käännöksenaikaiseen virheeseen.

Tämä ei tarkoita, että emme voisi epäsuorasti osoittaa vakio-oliota; se tarkoittaa vain, että se pitää tehdä esittelemällä osoitin, joka osoittaa vakio-oliota. Esimerkiksi:

```
const double *cptr;
```

cptr on osoitin double-tyyppiseen const-olioon. (Voimme lukea sen määrittelyn oikealta vasemmalle näin: “cptr on osoitin double-tyyppiseen olioon, joka on määritelty vakioksi (const)”.) Vaikeasti selitettävä asia on, että cptr itse ei ole vakio; voimme asettaa cptr:n osoittamaan eri osoitteeseen, mutta emme voi muokata oliota, jota se osoittaa. Esimerkiksi:

```
const double *pc = 0;
const double minWage = 9.60;

// ok: emme voi muuttaa minWage:a pc:n kautta
pc = &minWage;

double dval = 3.14;

// ok: ei voi muuttaa dval:ia pc:n kautta,
```

```
// vaikka dval itse ei ole vakio
pc = &dval; // ok

dval = 3.14159; // ok
*pc = 3.14159;  // virhe
```

Vakio-olion osoite voidaan sijoittaa vain vakio-olion osoittimeen kuten pc. Vakio-olion osoittimeen voidaan kuitenkin sijoittaa muuttujan osoite kuten tässä

```
pc = &dval;
```

Vaikka dval ei ole vakio, yritys muuttaa sen arvoa pc:n kautta johtaa silti käännöksenaikaiseen virheeseen (jälleen siksi, koska kääntäjä ei voi käytännössä päätellä todellista oliota, johon osoitin osoittaa kullakin hetkellä ohjelman suorituksen aikana).

Tosielämän ohjelmissa vakio-olion osoitinta käytetään useimmiten funktion muodollisena parametrina. Se toimii ikään kuin sopimuksena taaten, että funktiolle välitettyä todellista oliota ei muokata tuossa funktiossa. Esimerkiksi:

```
// tosielämän ohjelmissa vakioiden osoittimia
// käytetään useimmiten parametreina funktioihin
int strcmp( const char *str1, const char *str2 );
```

(Meillä on paljon lisää sanottavaa vakioiden osoittimista, kun käsittelemme funktioita luvussa 7.)

Voimme myös määritellä vakio-osoittimen joko vakio- tai muuttujaolioon. Esimerkiksi:

```
int errNumb = 0;
int *const curErr = &errNumb;
```

curErr on vakio-osoitin muuttujaolioon. (Voimme lukea tämän määrittelyn oikealta vasemmalle näin: "curErr on vakio-osoitin olioon, jonka tyyppi on int".) Tämä tarkoittaa, että emme voi sijoittaa curErr-vakio-osoittimeen toisen osoitteen arvoa, mutta voimme muokata arvoa, jota curErr osoittaa. Seuraavassa on esimerkki, kuinka voisimme käyttää curErr-vakio-osoitinta:

```
do_something();

if ( *curErr ) {
```

```
    errorHandler();
   *curErr = 0; // ok: alusta olio, jota osoitetaan
}
```

Yritys sijoittaa vakio-osoittimeen aiheuttaa käännöksenaikaisen virheen:

```
curErr = &myErrNumb; // virhe
```

Vakio-osoitin vakio-olioon määritellään yhdistämällä kaksi aikaisempaa määrittelyä. Esimerkiksi:

```
const double pi = 3.14159;
const double *const pi_ptr = &pi;
```

Tässä tapauksessa pi_ptr:n osoittamaa oliota eikä itse osoitetta voida muuttaa. (Voimme lukea tämän määrittelyn oikealta vasemmalle näin: "pi_ptr on vakio-osoitin olioon, jonka tyyppi on double ja määritelty vakioksi (const)".)

Harjoitus 3.16

Selitä seuraavien viiden määrittelyn tarkoitus. Yksilöi kaikki ne määrittelyt, jotka eivät ole sallittuja.

```
(a) int i;          (d) int *const cpi;
(b) const int ic;   (e) const int *const cpic;
(c) const int *pic;
```

Harjoitus 3.17

Mitkä seuraavista alustuksista ovat sallittuja? Selitä miksi.

```
(a) int i = -1;
(b) const int ic = i;
(c) const int *pic = &ic;
(d) int *const cpi = &ic;
(e) const int *const cpic = &ic;
```

Harjoitus 3.18

Aikaisemman harjoituksen määrittelyihin perustuen, mitkä seuraavista sijoituksista ovat sallittuja? Selitä miksi.

```
(a) i = ic;     (d) pic = cpic;
(b) pic = &ic;  (e) cpic = &ic;
(c) cpi = pic;  (f) ic = *cpic;
```

3.6 Viittaustyypit

Viittaus, jota joskus kutsutaan *aliakseksi*, toimii olion vaihtoehtoisena nimenä. Viittaus mahdollistaa olion epäsuoran käsittelyn samaan tapaan kuin osoitinta käytettäessä, mutta ei vaadi osoitinsyntaksin käyttöä. Tosielämän ohjelmissa viittauksia käytetään etupäässä muodollisina parametreina funktiolle — usein välittämään olioita funktioon. Tässä esityksessä haluamme kuitenkin esittää ja kuvata niiden käyttöä itsenäisinä olioina.

Viittaustyyppi määritellään, kuten seuraavassa, tyyppimääritteellä ja osoiteoperaattorilla. Viittaus pitää alustaa. Esimerkiksi:

```
int ival = 1024;

// ok: refVal on viittaus ival-olioon
int &refVal = ival;

// virhe: viittaus pitää alustaa olioon
int &refVal2;
```

Vaikka viittaus toimii eräänlaisena osoittimena, ei ole oikein alustaa sitä olion osoitteella kuten tekisimme osoittimella. Voimme kuitenkin määritellä osoitinviittauksen. Esimerkiksi:

```
int ival = 1024;

// virhe: refVal on tyyppiä int, ei int*
int &refVal = &ival;

int *pi = &ival;

// ok: refPtr on viittaus osoittimeen
int *&ptrVal2 = pi;
```

Kun viittaus on määritelty, sitä ei voida asettaa viittaamaan toiseen olioon (tässä on syy, miksi se pitää alustaa). Esimerkiksi seuraava sijoitus ei saa refVal:ia viittaamaan nyt min_val:iin. Sen sijaan se asettaa ival:in — olio, johon refVal viittaa — arvoksi min_val.

```
int min_val = 0;

// ival:in arvoksi on asetettu min_val:in arvo
// refVal:ia ei ole asetettu viittaamaan min_val:iin
refVal = min_val;
```

Kaikki viittausoperaatiot kohdistuvat todellisuudessa olioon, johon ne viittaavat osoiteoperaattori mukaan lukien. Esimerkiksi:

```
refVal += 2;
```

lisää arvon 2 ival:iin, joka on olio, johon refVal viittaa. Samalla tavalla

```
int ii = refVal;
```

sijoittaa ii:hin arvon, joka liittyy tuolla hetkellä ival:iin, kun taas

```
int *pi = &refVal;
```

alustaa pi:n ival:in osoitteella.

Jokaisen viittauksen määrittelyn edessä pitää olla osoiteoperaattori. (Tämä on sama aihe, jota käsittelimme aikaisemmin osoitinten yhteydessä.) Esimerkiksi:

```
// määrittelee kaksi int-tyyppistä oliota
int ival = 1024, ival2 = 2048;

// määrittelee viittauksen ja yhden olion
int &rval = ival, rval2 = ival2;

// määrittelee yhden olion, yhden osoittimen, yhden viittauksen
int ival3 = 1024, *pi = &ival3, &ri = ival3;

// määrittelee kaksi viittausta
int &rval3 = ival3, &rval4 = ival2;
```

const-viittaus voidaan alustaa toisentyyppisellä oliolla (edellyttäen, että on olemassa konversio tyypistä toiseen) kuten myös osoitteettomilla arvoilla kuten literaalivakioilla. Esimerkiksi:

```
double dval = 3.14159;

// sallittu vain const-viittauksille
const int &ir = 1024;
const int &ir2 = dval;
const double &dr = dval + 1.0;
```

Samanlaiset alustukset eivät ole sallittuja muille kuin const-viittauksille, vaan ne johtavat käännöksenaikaiseen virheeseen. Syy on hieman vaikea ymmärtää ja vaatii pienen selityksen.

Sisäisesti viittaus pitää yllä olion osoitetta, jolle se on alias. Kun on kyseessä osoitteettomia arvoja kuten literaalivakio ja erityyppisiä olioita, pitää kääntäjän generoida tilapäinen olio, johon viittaus varsinaisesti osoittaa, mutta johon käyttäjällä ei ole pääsyä. Kun esimerkiksi kirjoitamme

```
double dval = 1024;
const int &ri = dval;
```

kääntäjä muuntaa ne jotakuinkin näin:

```
int temp = dval;
const int &ri = temp;
```

Jos sijoittaisimme ri:hin uuden arvon, emme muuttaisi dval:ia, vaan sen sijaan temp:iä. Käyttäjälle se on sama, kuin että muutos ei yksinkertaisesti toimisi (eikä toimi kaikesta siitä hyvästä huolimatta, jota se tekee käyttäjälle).

const-viittaukset eivät tuo esille tätä ongelmaa, koska ne ovat vain lukua varten. Kun ei sallita ei-const-viittauksia olioihin tai arvoihin, jotka vaativat tilapäisolioita, näyttää se yleensä paremmalta ratkaisulta kuin sallia viittauksen määrittely, joka ei näytä toimivan.

Tässä on esimerkki, joka on vaikea esitellä oikein ensimmäisellä kerralla. Haluamme alustaa viittauksen const-olion osoitteella. const-määritteetön viittaus ei ole sallittu ja se saa aikaan käännöksenaikaisen virheen:

```
const int ival = 1024;

// virhe: vaatii const-viittauksen
int *&pi_ref = &ival;
```

Ensimmäinen yrityksemme korjata pi_ref:in määrittely voisi olla seuraava, mutta se ei toimi — huomaatko, miksi?

```
const int ival = 1024;

// yhä virhe
const int *&pi_ref = &ival;
```

Jos luemme tämän määrittelyn oikealta vasemmalle, huomaamme, että pi_ref on viittaus osoittimeen olioon, joka on tyyppiä int ja määritelty olemaan vakio (const). Viittauksemme ei ole vakioon, vaan sen sijaan muuhun kuin vakio-osoittimeen, joka osoittaa vakio-oliota. Oikea määrittely on seuraavassa:

```
const int ival = 1024;

// ok: tämä on sallittua
int *const &pi_ref = &ival;
```

Kaksi pääeroa viittauksen ja osoittimen välillä on: viittauksen pitää aina viitata olioon ja yhden viittauksen sijoitus toiseen muuttaa olita, johon viitataan, eikä viittausta itseään. Katsotaanpa esimerkkiä. Kun kirjoitamme

```
int *pi = 0;
```

alustamme pi:n arvolla 0 — pi ei juuri nyt osoita oliota. Kuitenkin, kun kirjoitamme

```
const int &ri = 0;
```

niin muista, että sisäisesti tapahtuu seuraava muunnos:

```
int temp = 0;
const int &ri = temp;
```

Yhden viittauksen sijoitus toiseen on toinen ero. Kun on kirjoitettu seuraavasti:

```
int ival = 1024, ival2 = 2048;
int *pi = &ival, *pi2 = &ival2;
```

kirjoitamme

```
pi = pi2;
```

ival — olio, johon pi osoittaa — pysyy muuttumattomana. pi:hin sen sijaan sijoitetaan olio, johon pi2 osoittaa — tässä tapauksessa ival2. Merkittävää on se, että pi ja pi2 molemmat osoittavat nyt samaa oliota. (Tämä voi olla merkittävä ohjelmavirheen lähde, jos kopioimme yhden luokan toiseen, jossa yksi tai useampi jäsen ovat osoittimia. Katsomme tätä ongelmaa yksityiskohtaisesti luvussa 14.)

Kuitenkin, kun on annettu seuraava

```
int &ri = ival, &ri2 = ival2;
```

ja kirjoitamme

```
ri = ri2;
```

se, joka muuttuu, on ival-arvo, johon ri viittaa, eikä viittaus. Sijoituksen jälkeen nuo kaksi yhä viittaavat alkuperäisiin olioihinsa.

Tosielämän C++-ohjelmat käyttävät harvemmin itsenäisiä viittaustyyppisiä olioita. Sen sijaan viittauksia käytetään pääasiassa funktioiden muodollisina parametreina. Esimerkiksi:

```
// tosielämän esimerkki siitä, kuinka viittauksia käytetään

// palauta käsittelyn tila. Sijoita arvo parametriin
bool get_next_value( int &next_value );

// ylikuormitettu lisäysoperaattori
Matrix operator+( const Matrix&, const Matrix& );
```

Kuinka tämä viittauksen käyttö liittyy itsenäisten, viittaustyyppisten olioiden käsittelyyn? Käynnistyksessä kuten tässä

```
int ival;
while ( get_next_value( ival )) ...
```

todellisen argumentin, joka tässä tapauksessa on ival, sitominen muodollisella parametrilla next_value, on yhtäpitävä seuraavan itsenäisen määrittelyn kanssa:

```
int &next_value = ival;
```

(Viittauksien käyttö funktion parametreina käsitellään yksityiskohtaisesti luvussa 7.)

Harjoitus 3.19

Mitkä seuraavista määrittelyistä eivät ole kelvollisia, vai onko sellaisia? Miksi? Kuinka korjaisit ne?

```
(a) int ival = 1.01;       (b) int &rval1 = 1.01;
(c) int &rval2 = ival;     (d) int &rval3 = &ival;
(e) int *pi = &ival;       (f) int &rval4 = pi;
(g) int &rval5 = *pi;      (h) int &*prval1 = pi;
(i) const int &ival2 = 1;  (j) const int &*prval2 = &ival;
```

Harjoitus 3.20

Kun on tehty seuraavat määrittelyt, mitkä niistä eivät ole kelvollisia, vai onko yksikään?

```
(a) rval1 = 3.14159;
(b) prval1 = prval2;
(c) prval2 = rval1;
(d) *prval2 = ival2;
```

Harjoitus 3.21

Mitä eroja on määrittelyiden (a) ja sijoitusten (b) välillä? Mitkä niistä eivät ole kelvollisia, vai onko yksikään?

```
(a) int ival = 0;
    const int *pi = 0;
    const int &ri = 0;

(b) pi = &ival;
    ri = &ival;
    pi = &rval;
```

3.7 Bool-tyyppi

Bool-olioon voidaan sijoittaa literaaliarvot true ja false. Esimerkiksi:

```
// alusta merkkijono etsittävällä sanalla
string search_word = get_word();

// alusta bool-muuttuja arvolla false
bool found = false;

string next_word;
while ( cin >> next_word )
      if ( next_word == search_word )
       found = true;
// ...

// lyhennetty merkintätapa tälle: if ( found == true )
if ( found )
     cout << "ok, löysimme sanan\n";
 else cout << "ehei, sanaa ei löytynyt.\n";
```

Vaikka bool-olio voi pitää arvonaan jommankumman kokonaistyypeistä, ei sitä voida esitellä tyypiksi signed, unsigned, short tai long. Esimerkiksi seuraava ei ole sallittua:

```
// virhe: emme voi määrittää bool-tyyppiä short-tyyppiseksi
short bool found = false;
```

Sekä bool-olio, kuten found, että bool-literaalit “ylennetään” (*to promote*) implisiittisesti int-tyyppisiksi arvoiksi tarvittaessa (kuten seruaavassa esimerkissä): false-arvosta tulee 0 ja true-arvosta tulee 1. Esimerkiksi:

```
bool found = false;
int occurrence_count = 0;

while ( /* muminaa...  */ )
{
    found = look_for( /* jotain... */ );

    // found-arvo ylennetään joko arvoon 0 tai 1
    occurrence_count += found;
}
```

Aivan kuten false- ja true-literaalit ylennetään automaattisesti tarvittaessa arvoihin 0 ja 1 myös aritmeettiset ja osoitinarvot konvertoidaan implisiittisesti bool-tyypin arvoisiksi. Nolla-arvo tai osoittimen null-arvo konvertoidaan false-arvoiksi ja kaikki muut konvertoidaan true-arvoiksi. Esimerkiksi:

```
// palauttaa esiintymien lukumäärän
extern int find( const string& );
bool found = false;
if ( found = find( "ruusunnuppu" ))
        // ok: found == true

// palauttaa osoittimen esiintymään, jos löytyy
extern int* find( int value );

if ( found = find( 1024 ))
        // ok: found == true
```

3.8 Luetellun joukon tyypit

Usein ohjelmoidessamme meidän pitää määritellä joukko vaihtoehtoisia ominaisuuksia, jotka liittyvät johonkin olioon. Esimerkiksi tiedosto voidaan avata jossain kolmessa eri tilassa: input, output ja append.

Tyypillisesti hoidamme tällaiset tila-arvot ja ominaisuudet liittämällä niihin kaikkiin yksilölliset vakionumerot. Täten voisimme kirjoittaa seuraavasti:

```
const int input = 1;
const int output = 2;
const int append = 3;
```

ja käyttää näitä vakioita seuraavasti:

```
bool open_file( string file_name, int open_mode);

// ...
open_file( "Turku_ja_Pori", append );
```

Vaikka tämä toimii, siinä on monia heikkouksia. Suurin niistä on, että ei ole olemassa tapaa rajoittaa välitettyjä arvoalueita arvoille input, output ja append.

Luetellut joukot tarjoavat vaihtoehtoisen metodin, eivät vain määrittelemällä, vaan myös ryhmittelemällä joukon kokonaisvakioita. Esimerkiksi:

```
enum open_modes{ input = 1, output, append };
```

open_modes on luetellun joukon tyyppi. Jokainen nimetty jäsen määrittelee yksilöllisen tyypin ja sitä voidaan käyttää tyyppimääreenä. Esimerkiksi:

```
void open_file( string file_name, open_modes om );
```

input, output ja append on *lueteltu joukko* (*enumerators*). Ne edustavat sitä täydellistä joukkoa arvoja, joilla open_modes voidaan alustaa tai sijoittaa. Esimerkiksi:

```
open_file( "Turku_ja_Pori", append );
```

Jos yritämme välittää open_file()-funktiolle minkä tahansa muun arvon kuin input, output tai append, tapahtuu käännöksenaikainen virhe. Lisäksi, jos yritämme välittää sille yhtä suuren kokonaisarvon kuten tässä, antaa kääntäjä yhä virheilmoituksen:

```
// virhe: 1 ei ole open_modes:in luetellun joukon jäsen  ...
open_file( "Joonas", 1 );
```

Lisäksi voimme esitellä luetellun joukon tyyppisiä oliota kuten tässä:

```
open_modes om = input;
// ...
om = append;
```

ja käyttää om-oliota luetellun joukon jäsenen tilalla:

```
open_file( "JälkiPuhe", om );
```

Yksi asia, jota emme voi tehdä luetellulla joukolla, on tulostaa varsinaisia nimiä. Kun kirjoitamme

```
cout << input << " " << om << endl;
```

tulostuu

```
1 3
```

Eräs ratkaisu on määritellä merkkijonotaulukko, jota indeksoidaan luetellun joukon arvoilla. Siten voisimme kirjoittaa

```
cout << open_modes_table[ input ] << " "
    << open_modes_table[ om ]    << endl;
```

ja generoida

```
input append
```

Toinen asia, jota emme voi tehdä, on luetellun joukon arvojen iterointi kuten tässä:

```
// ei tueta
for ( open_modes iter = input; iter != append; ++iter )
    // ...
```

Tukea ei ole olemassa siirtymiselle eteen- ja taaksepäin luetellun joukon arvosta toiseen.

Lueteltu joukko määritellään avainsanalla enum, joka sen valinnainen nimi, ja aaltosulkujen sisällä olevalla, pilkuin erotellulla luettelolla sen jäsenistä. Oletusarvo on, että luetellun joukon ensimmäinen jäsen saa arvon 0. Sitä seuraaviin jäseniin sijoitetaan arvo, joka on 1:tä suurempi kuin edeltävän arvo. Esimerkissämme sijoitamme input:in arvoksi 1. output saa automaattisesti arvokseen 2 ja append arvon 3. Seuraavassa luetellussa joukossa saa jäsen shape arvon 0, sphere arvon 1, cylinder arvon 2 ja polygon arvon 3.

```
// shape == 0, sphere == 1, cylinder == 2, polygon == 3
enum Forms{ shape, sphere, cylinder, polygon };
```

Luetellun joukon jäseneen voidaan sijoittaa eksplisiittisesti arvo. Tämän arvon ei tarvitse olla yksilöllinen. Seuraavassa esimerkissä point2d saa arvon 2, point2w:tä kasvatetaan yhdellä arvoon 3, point3d saa eksplisiittisesti arvokseen 3 ja point3w jälleen oletusarvoisesti kasvatetaan yhdellä arvoon 4.

```
// point2d == 2, point2w == 3, point3d == 3, point3w == 4
enum Points { point2d = 2, point2w, point3d = 3, point3w };
```

Luetellun joukon tyyppisiä olioita voidaan määritellä, laittaa mukaan lausekkeisiin ja välittää argumentteina funktioille. Luetellun joukon tyyppinen olio voidaan alustaa ja siihen sijoittaa vain toisella samanlaisella luetellun joukon tyypillä tai yhdellä sen jäsenistä. Vaikka esimerkiksi 3 on sallittu arvo, joka liittyy lueteltuun joukkoon Points, ei sitä voida eksplisiittisesti sijoittaa Points-olioon:

```
void mumble() {
   Points pt3d = point3d; // ok: pt3d == 3

   // virhe: pt2w alustettu int-tyypillä
   Points pt2w = 3;

   // virhe: polygon ei ole luetellun joukon Points jäsen
   pt2w = polygon;

   // ok: molemmat ovat luetellun tyypin, Points, olioita
   pt2w = pt3d;
}
```

Kuitenkin tarvittaessa luetellun joukon tyyppi ylennetään automaattisesti aritmeettiseksi tyypiksi. Esimerkiksi:

```
const int array_size = 1024;

// ok: pt2w ylennetään int-tyyppiseksi
int chunk_size = array_size * pt2w;
```

3.9 Taulukkotyypit

Kuten näimme kohdassa 2.1, on taulukko kokoelma yhdentyyppisiä olioita. Yksittäisiä olioita ei nimetä, vaan sen sijaan jokaista käsitellään positiossaan taulukossa. Tätä käsittelymuotoa kutsutaan *indeksoinniksi*. Esimerkiksi

```
int ival;
```

esittelee yksittäisen kokonaislukuolion, kun taas

```
int ia[ 10 ];
```

esittelee kymmenen kokonaislukuolion taulukon. Jokaista oliota kutsutaan ia:n *elementiksi*. Täten

```
ival = ia[ 2 ];
```

sijoittaa ival-olioon arvon, joka on tallennettu ia:n elementtiin indeksillä 2. Samalla tavalla

```
ia[ 7 ] = ival;
```

sijoittaa ival-olion arvoksi ia:n elementin indeksillä 7.

Taulukon määrittely muodostuu tyyppimääritteestä, tunnuksesta ja ulottuvuudesta. Ulottuvuus, joka määrittää taulukon elementtien lukumäärän, suljetaan hakasulkuparin sisään. Taulukon ulottuvuudeksi on annettava suurempi tai yhtäsuuri kuin 1. Ulottuvuuden arvon pitää olla vakiolauseke — eli sen pitää olla tiedossa käännöksen aikana. Tämä tarkoittaa, että muita kuin const-muuttujaa ei voida määrittää taulukon ulottuvuudeksi. Seuraavat ovat esimerkkejä sekä sallituista että kelpaamattomista taulukon määrittelyistä:

```
extern int get_size();

// sekä buf_size että max_files ovat vakioita
const int buf_size = 512, max_files = 20;
int staff_size = 27;

// ok: const-muuttuja
char input_buffer[ buf_size ];

// ok: constant-lauseke: 20 - 3
char *fileTable[ max_files - 3 ];

// virhe: muu kuin const-muuttuja
double salaries[ staff_size ];

// virhe: muu kuin const-lauseke
int test_scores[ get_size() ];
```

Vaikka staff_size alustetaan literaalivakiolla, itse staff_size on muu kuin const-olio. Sen arvo selviää vasta suorituksen aikana, joten se ei kelpaa taulukon ulottuvuudeksi. Toisaalta lauseke

```
max_files - 3
```

on vakiolauseke, koska max_files on const-muuttuja alkuarvolla 20. Tämän arvoksi lasketaan käännöksen aikana 17.

Kuten näimme kohdassa 2.1, taulukon elementit numeroidaan alkaen arvosta 0. Taulukon, jossa on kymmenen elementtiä, indeksien oikeat arvot ovat väliltä 0 — 9 eikä väliltä 1 — 10. Seuraavassa esimerkissä for-silmukka askeltaa läpi taulukon kymmenen elementtiä ja alustaa jokaisen indeksinsä arvolla:

```
int main()
{
   const int array_size = 10;
   int ia[ array_size ];

   for ( int ix = 0; ix < array_size; ++ix )
        ia[ ix ] = ix;
}
```

Taulukko voidaan alustaa eksplisiittisesti pilkuin erotellulla luettelolla arvoja, jotka ovat aaltosulkujen sisällä. Esimerkiksi:

```
const int array_size = 3;
int ia[ array_size ] = { 0, 1, 2 };
```

Eksplisiittisesti alustetulle taulukolle ei tarvitse määrittää ulottuvuuden arvoa. Kääntäjä päättelee taulukon koon lueteltujen elementtien lukumäärästä:

```
// taulukko, jonka ulottuvuus on 3 elementtiä
int ia[] = { 0, 1, 2 };
```

Jos ulottuvuuden koko on määritetty, ei elementtien lukumäärä saa ylittää tuota arvoa. Muussa tapauksessa tapahtuu käännöksenaikainen virhe. Jos ulottuvuuden koko on suurempi kuin lueteltujen elementtien lukumäärä, taulukon eksplisiittisesti alustamattomien elementtien arvoksi asetetaan 0.

```
// ia ==> { 0, 1, 2, 0, 0 }
const int array_size = 5;
int ia[ array_size ] = { 0, 1, 2 };
```

Merkkitaulukko voidaan alustaa joko pilkuin erotellulla merkkiliteraalijoukolla aaltosuluissa tai merkkijonoliteraalilla. Huomaa kuitenkin, että nämä kaksi muotoa eivät ole sama asia. Merkkijonovakio sisältää lisäksi päättävän null-merkin. Esimerkiksi:

```
const char ca1[] = { 'C', '+', '+' };
const char ca2[] = "C++";
```

ca1:n ulottuvuus no 3; ca2:n ulottuvuus on 4. Seuraava esittely aiheuttaa käännöksenaikaisen virheen:

```
// virhe: "Daniel" on 7 elementtiä
const char ch3[ 6 ] = "Daniel";
```

Taulukkoa ei voi alustaa toisella taulukolla eikä yhtä taulukkoa voi sijoittaa toiseen. Lisäksi ei ole sallittua esitellä viittauksien taulukkoa.

```
const int array_size = 3;
int ix, jx, kx;

// ok: int*-tyyppisten osoittimien taulukko
int *iap [] = { &ix, &jx, &kx };

// virhe: viittausten taulukkoa ei sallita
int &iar[] = { ix, jx, kx };

// virhe: ei voi alustaa yhtä taulukkoa toisella
int ia2[] = ia; // error

int main()
{
   int ia3[ array_size ]; // ok

   // virhe: ei voi sijoittaa taulukkoa toiseen
   ia3 = ia;
   return 0;
}
```

Jotta taulukko voidaan kopioida toiseen, pitää jokainen elementti kopioida vuorollaan. Esimerkiksi:

```
const int array_size = 7;
int ia1[] = { 0, 1, 2, 3, 4, 5, 6 };

int main()
{
   int ia2[ array_size ];

   for ( int ix = 0; ix < array_size; ++ix )
        ia2[ ix ] = ia1[ ix ];

   return 0;
}
```

Jokaista lauseketta, jonka tuloksena on kokonaisarvo, voidaan käyttää taulukon indeksinä. Esimerkiksi:

```
int someVal, get_index();
ia2[ get_index() ] = someVal;
```

Käyttäjien tulisi kuitenkin olla tietoisia, että kielellä ei ole käännöksen- tai suorituksenaikaisia raja-arvotarkistuksia indeksille. Mikään ei estä ohjelmoijaa askeltamasta taulukon rajojen yli, ellei hän kiinnitä huomioita yksityiskohtiin ja testaa ohjelmaa läpikotaisin. Ei ole tavatonta, että ohjelma kääntyy ja lähtee suoritukseen, mutta on silti pahemman kerran virheellinen.

Harjoitus 3.22

Mitkä seuraavista taulukon määrittelyistä eivät ole sallittuja? Selitä miksi.

```
int get_size();
int buf_size = 1024;

(a) int ia[ buf_size ];     (d) int ia[ 2 * 7 - 14 ];
(b) int ia[ get_size() ];   (e) char st[ 11 ] = "fundamental";
(c) int ia[ 4 * 7 - 14 ];
```

Harjoitus 3.23

Tämä koodikatkelma aikoo alustaa taulukon jokaisen elementin indeksinsä arvolla. Se sisältää kuitenkin lukuisia indeksointivirheitä. Yksilöi ne.

```
int main() {
    const int array_size = 10;
    int ia[ array_size ];

    for ( int ix = 1; ix <= array_size; ++ix )
        ia[ ix ] = ix;

    // ...
}
```

3.9.1 Moniulotteiset taulukot

Myös moniulotteisia taulukoita voidaan määritellä. Jokainen ulottuvuus määritetään omalla hakasulkuparilla. Esimerkiksi

```
int ia[ 4 ][ 3 ];
```

määrittelee kaksiulotteisen taulukon. Ensimmäistä ulottuvuutta kutsutaan *riviksi*, toista ulottuvuutta *sarakkeeksi*. ia on kaksiulotteinen taulukko, jossa on neljä riviä, kolme elementtiä kussakin. Myös moniulotteisia taulukoita voidaan alustaa.

```
int ia[ 4 ][ 3 ] = {
    { 0, 1, 2 },
    { 3, 4, 5 },
    { 6, 7, 8 },
    { 9, 10, 11 }
};
```

Sisäkkäiset hakasulut, jotka ilmaisevat rivejä, ovat valinnaisia. Seuraava alustus on yhtäpitävä edellisen kanssa, vaikkakin paljon sekavampi.

```
int ia[4][3] = { 0,1,2,3,4,5,6,7,8,9,10,11 };
```

Seuraava määrittely alustaa jokaisen rivin ensimmäisen elementin. Jäljelle jäävät elementit alustetaan arvoon 0.

```
int ia[ 4 ][ 3 ] = { {0}, {3}, {6}, {9} };
```

Jos sisäkkäiset hakasulut jätetään pois, tulokset voivat olla erilaisia. Seuraava määrittely

```
int ia[ 4 ][ 3 ] = { 0, 3, 6, 9 };
```

alustaa ensimmäisen rivin kolme ensimmäistä elementtiä ja toisen rivin ensimmäisen elementin. Jäljelle jäävät elementit alustetaan arvoon 0. Moniulotteisen taulukon indeksointi vaatii hakasulkuparin jokaiselle ulottuvuudelle. Esimerkiksi seuraavat sisäkkäiset for-silmukat alustavat kaksiulotteisen taulukon:

```
int main()
{
   const int rowSize = 4;
   const int colSize = 3;
   int ia[ rowSize ][ colSize ];

   for ( int i = 0; i < rowSize; ++i )
        for ( int j = 0; j < colSize; ++j )
            ia[ i ][ j ] = i + j;
}
```

Vaikka lauseke

```
ia[ 1, 2 ]
```

on sallittu rakenne C++:ssa, ei ohjelmoija todennäköisesti ole tarkoittanut ilmaista sitä: `ia[1,2]` on yhtä kuin `ia[2]`, koska `1,2` tulkitaan pilkkulausekkeeksi, joka johtaa yksittäiseen arvoon 2 (pilkkulauseke käsitellään kohdassa 4.10). Tämä käsittelee `ia`:n kolmannen rivin ensimmäistä elementtiä. Ohjelmoija on varmaan tarkoittanut `ia[1][2]`.

C++:ssa moniulotteinen indeksointi vaatii erillisen hakasulkuparin jokaiselle indeksille, jota ohjelmoija haluaa käsitellä.

3.9.2 Taulukko- ja osoitintyyppien keskinäinen suhde

Olkoon annettu seuraava määrittely

```
int ia[] = { 0, 1, 1, 2, 3, 5, 8, 13, 21 };
```

Mitä tarkoitetaan, kun kirjoitetaan yksinkertaisesti

```
ia;
```

Taulukon tunnus tarkoittaa sen sisältämän ensimmäisen elementin osoitetta. Se on osoitintyyppi taulukon sisältämään elementtiin. ia:n tapauksessa sen tyyppi on int*. Joten seuraavat kaksi muotoa ovat täsmälleen samanarvoiset ja ne palauttavat taulukon ensimmäisen elementin osoitteen:

```
ia;
&ia[0];
```

Samalla tavalla, jos käsittelemme arvoa, voimme kirjoittaa kummalla tahansa tavalla seuraavista:

```
// molemmat johtavat ensimmäisen elementin arvoon
*ia;
ia[0];
```

Tiedämme, kuinka käsitellä toisen elementin osoitetta indeksioperaattoria käyttäen:

```
&ia[1];
```

Siitä seuraa, että seuraava

```
ia+1;
```

myös johtaa toisen elementin osoitteeseen jne. Samalla tavalla molemmat seuraavista käsittelevät toisen elementin arvoa:

```
*(ia+1);
ia[1];
```

On aivan eri asia kirjoittaa

```
*ia+1;
```

kuin

```
*(ia+1);
```

Käänteisoperaattorilla on korkeampi sidontajärjestys kuin lisäysoperaattorilla (käsittelemme sidontajärjestystä kohdassa 4.13) ja siten se käsitellään ensiksi. ia:n käänteinen osoitus palauttaa taulukon ensimmäisen elementin arvon. Siihen lisätään sen jälkeen 1. Kun sijoitetaan sulut lausekkeen ympärille, ia:han lisätään ensiksi 1 ja sitten osoitetaan uutta osoitetta käänteisesti. Kun ia:han lisätään 1, se kasvattaa ia:ta vastaavan elementin koolla; ia+1 osoittaa taulukon seuraavaan elementtiin.

Taulukon läpikäynti voidaan siten toteuttaa joko indeksoimalla, kuten olemme tehneet tähän saakka, tai suoran osoitinkäsittelyn kautta. Esimerkiksi:

```
#include <iostream>
int main()
{
   int ia[9] = { 0, 1, 1, 2, 3, 5, 8, 13, 21 };
   int *pbegin = ia;
   int *pend = ia + 9;
```

```
    while ( pbegin != pend ) {
          cout << *pbegin << ' ';
          ++pbegin;
    }
}
```

pbegin alustetaan taulukon ensimmäisen elementin osoitteella. Sitä kasvatetaan jokaisella while-silmukan toistokerralla seuraavan elementin osoitteeseen. Vaikea tehtävä on päättää, milloin lopettaa. Esimerkissämme alustamme pend:in osoittamaan yhden elementin yli taulukon viimeisen elementin. Kun pbegin on yhtä suuri kuin pend, tiedämme, että olemme silmukoineet läpi koko taulukon.

Jos asetamme osoitinparin osoittamaan taulukon alkuun ja yhden sen yli erilliseen funktioon, silloin meillä on keino käydä taulukko läpi tietämättä sen todellista kokoa (vaikkakin ohjelmoijan, joka funktion käynnistää, pitää tietää se). Esimerkiksi:

```
#include <iostream>

void ia_print( int *pbegin, int *pend )
{
      while ( pbegin != pend ) {
            cout << *pbegin << ' ';
            ++pbegin;
      }
}

int main()
{
   int ia[9] = { 0, 1, 1, 2, 3, 5, 8, 13, 21 };
   ia_print( ia, ia + 9 );

}
```

Tämä on tietysti yhä rajoittunut: sen tuki on rajoittunut osoittamaan taulukoihin, joiden tyyppi on int. Voimme vaikuttaa rajoitukseen tekemällä ia_print()-funktiosta mallifunktion (käsittelimme lyhyesti malleja kohdassa 2.5). Esimerkiksi:

```
#include <iostream>

template <class elemType>
void print( elemType *pbegin, elemType *pend )
{
      while ( pbegin != pend ) {
            cout << *pbegin << ' ';
            ++pbegin;
      }
}
```

Nyt voimme välittää geneeriselle print()-funktiollemme osoitinparin minkä tahansa tyyppiseen taulukkoon, jolle tulostusoperaattori on määritelty. Esimerkiksi:

```
int main()
{
   int ia[9] = { 0, 1, 1, 2, 3, 5, 8, 13, 21 };
   double da[4] = { 3.14, 6.28, 12.56, 25.12 };
   string sa[3] = { "piglet", "eeyore", "pooh" };

   print( ia, ia+9 );
   print( da, da+4 );
   print( sa, sa+3 );
}
```

Tämänkaltaista ohjelmointia sanotaan *geneeriseksi ohjelmoinniksi*. Vakiokirjastossa on kokoelma geneerisiä algoritmeja (näimme niitä lyhyesti kohdassa 2.8 ja kohdan 3.4 lopussa), joille annetaan alku/loppu-osoitinpari ilmaisemaan läpikäytävien elementtien väliä. Esimerkiksi geneerinen sort()-algoritmi voidaan käynnistää kuten seuraavassa:

```
#include <algorithm>
int main()
{
   int ia[6] = { 107, 28, 3, 47, 104, 76 };
   string sa[3] = { "piglet", "eeyore", "pooh" };

   sort( ia, ia+6 );
   sort( sa, sa+3 );
}
```

Geneerisiä algoritmeja käsitellään yksityiskohtaisesti luvussa 12. Kirjan liitteessä ne on lueteltuna aakkosjärjestyksessä käyttöesimerkkeineen.

Yleisemmin sanottuna, vakiokirjastossa on joukko luokkia, jotka kapseloivat säiliöiden ja osoittimien abstraktioita. Esittelimme molemmat lyhyesti kohdassa 2.8. Seuraavassa katsomme vektorin säiliötyyppiä, joka on oliopohjainen vaihtoehto sisäiselle taulukolle.

3.10 Vektori-säiliötyyppi

Vektoriluokka (*vector*) on vaihtoehto sisäiselle taulukolle (esittelimme vektoriluokan lyhyesti kohdassa 2.8) ja yleensäkin suosittelemme sen käyttöä. (On silti monia ohjelmatilanteita, joissa sisäisen taulukon käyttö on välttämätöntä, kuten komentorivin valitsimien käsittely — katsomme sitä kohdassa 7.8). Vektoriluokka, kuten merkkijonoluokkakin, on osa vakiokirjastoa, joka kuuluu C++-standardiin.

Kun vektoria halutaan käyttää, pitää ottaa mukaan siihen liittyvä otsikkotiedosto:

```
#include <vector>
```

On olemassa kaksi keskenään täysin erilaista muotoa käyttää vektoria. Kutsumme muotoja *taulukkoidiomiksi* ja *STL-idiomiksi*. Taulukkoidiomissa matkimme sisäisen taulukon käyttöä: määrittelemme annetun kokoisen vektorin.

```
vector< int > ivec( 10 );
```

Tämä on sama, kuin jos määriteltäisiin sisäinen kymmenen elementin taulukko, kuten seuraavassa:

```
int ia[ 10 ];
```

Indeksioperaattoria voidaan käyttää vektorin elementtien käsittelyyn samalla tavalla, kuin käsittelisimme sisäisen taulukon elementtejä. Esimerkiksi:

```
void simple_example()
{
  const int elem_size = 10;
  vector< int > ivec( elem_size );
  int ia[ elem_size ];

  for ( int ix = 0; ix < elem_size; ++ix )
      ia[ ix ] = ivec[ ix ];

    // ...
}
```

Voimme kysellä vektorin kokoa (size()) tai testata, onko se tyhjä (empty()). Esimerkiksi:

```
void print_vector( vector<int> ivec )
{
      if ( ivec.empty() )
            return;

      for ( int ix = 0; ix < ivec.size(); ++ix )
            cout << ivec[ ix ] << ' ';
}
```

Vektorin elementit alustetaan elementtien tyypin oletusarvolla. Aritmeettisen ja osoitintyypin oletusarvo on 0. Oletusarvo saadaan luokan tyypille kutsumalla oletusmuodostajaa (katso kohdasta 2.3 oletusmuodostajan esittely). Vaihtoehtoisesti voimme antaa eksplisiittisen alkuarvon jokaiselle elementille. Esimerkiksi

```
vector< int > ivec( 10, -1 );
```

määrittelee niin, että ivec sisältää kymmenen int-tyyppistä elementtiä ja jokainen alustetaan arvolla –1.

Sisäisen taulukon elementit voimme eksplisiittisesti alustaa vakioarvojoukolla. Esimerkiksi:

```
int ia[ 6 ] = { -2, -1, 0, 1, 2, 1024 };
```

Emme voi alustaa vektorioliota eksplisiittisesti samalla tavalla. Voimme kuitenkin alustaa sen kokonaan tai osittain olemassaolevalla taulukolla määrittämällä osoitteen sen alkuun ja *yhden yli* sen viimeisen elementin, jolla haluamme vektorin alustettavan. Esimerkiksi:

```
// kopioi 6 elementtiä ia:sta ivec:iin
vector< int > ivec( ia, ia+6 );
```

Vektorille ivec välitetyt kaksi osoitinta merkitsevät arvoaluetta, jolla olio alustetaan. Toinen

osoitin osoittaa aina yhden kopioitavan elementin yli. Elementtien alue voi olla myös taulukon alijoukko. Esimerkiksi:

```
// kopioi 3 elementtiä: ia[2], ia[3], ia[4]
vector< int > ivec( &ia[ 2 ], &ia[ 5 ] );
```

Toisin kuin sisäinen taulukko, vektori voidaan alustaa tai siihen voidaan sijoittaa toinen vektori. Esimerkiksi:

```
vector< string > svec;

void init_and_assign()
{
        // alustetaan vektori toisella
        vector< string > user_names( svec );
        // ...

        // kopioidaan vektori toiseen
        svec = user_names;
}
```

STL-idiomissa[2] vektoria käytetään melko eri tavalla. Sen sijaan, että määriteltäisiin vektori annetulla koolla, määrittelemme sen tyhjänä:

```
vector< string > text;
```

Sen sijaan, että indeksoisimme ja sijoittaisimme elementtiin, lisäämme elementin vektoriin. Esimerkiksi push_back()-operaatio lisää elementin vektorin loppuun. Seuraava while-silmukka lukee peräkkäin merkkijonoja vakiosyötöstä lisäten jokaisen kerrallaan vektoriin:

```
string word;
while ( cin >> word ) {
        text.push_back( word );
        // ...
}
```

Vaikka voimme käydä elementit läpi indeksioperaattoria käyttäen:

```
cout << "luetut sanat ovat: \n";
for ( int ix = 0; ix < text.size(); ++ix )
    cout << text[ ix ] << ' ';
cout << endl;
```

käytämme tyypillisemmin iteraattoriparia. Sen palauttavat vektorin begin()- ja end()-operaatiot:

```
cout << "luetut sanat ovat: \n";
for ( vector<string>::iterator it = text.begin();
    it != text.end(); ++it )
```

2. STL on lyhenne sanoista Standard Template Library. Ennen kuin vektoriluokka tuli mukaan C++-standardiin yhdessä geneeristen algoritmien kanssa, se oli osa itsenäistä kirjastoa nimeltään STL ([katso MUSSER96]).

```
        cout << *it << ' ';
    cout << endl;
```

Iteraattori on vakiokirjaston luokka, joka edustaa osoittimen toimintoja. Ilmaisu

```
*it;
```

käyttää iteraattoria käänteisesti ja käsittelee todellista osoitettua oliota.

```
++it;
```

vie iteraattoria eteenpäin seuraavan elementin osoitteeseen. (Katsomme iteraattoreita, vektoreita ja yleistä STL-idiomia huomattavan yksityiskohtaisesti luvussa 6.)

Eräs varoitus koskee näiden kahden idiomin sotkemista keskenään. Esimerkiksi määrittelyn

```
vector<int> ivec;
```

tuloksena on tyhjä vektori. Jos kirjoitetaan

```
ivec[0] = 1024;
```

se on virhe, koska ensimmäistä elementtiä ei vielä ole. Voimme indeksoida vain elementtejä, jotka ovat jo vektorissa. size()-operaatio palauttaa vektorin sisältämien elementtien lukumäärän.

Samalla tavalla, kuin määrittelemme annetun kokoisen vektorin, kuten tässä:

```
vector<int> ia( 10 );
```

kaikki lisäykset kasvattavat kokoa sen sijaan, että kirjoitettaisiin olemassaolevien elementtien päälle. Se voi kuulostaa ilmeiseltä; kuitenkaan seuraava ei ole epätavallinen virhe aloittelijalle:

```
const int size = 7;
int ia[ size ] = { 0, 1, 1, 2, 3, 5, 8 };
vector< int > ivec( size );
for ( int ix = 0; ix < size; ++ix )
    ivec.push_back( ia[ ix ]);
```

Ohjelmoija päätyy siihen, että ivec sisältää 14 elementtiä, jotka ovat ia:n elementtejä kahdeksannesta elementistä alkaen.

Lisäksi STL-idiomissa voidaan yksi tai useampi vektorin elementti poistaa. (Jälleen, katsomme tätä ja kuvaamme sen käyttöä luvussa 6.)

Harjoitus 3.24

Mitkä seuraavista vektorien määrittelyistä ovat virheellisiä, vai onko yksikään?

```
int ia[ 7 ] = { 0, 1, 1, 2, 3, 5, 8 };

(a) vector< vector< int > > ivec;
(b) vector< int >    ivec = { 0, 1, 1, 2, 3, 5, 8 };
(c) vector< int >    ivec( ia, ia+7 );
(d) vector< string > svec = ivec;
```

```
(e) vector< string > svec( 10, string( "null" ));
```

Harjoitus 3.25

Kun on annettu seuraava funktion määrittely

```
bool is_equal( const int*ia, int ia_size,
        const vector<int> &ivec );
```

toteuta seuraava käyttäytyminen: jos kaksi säiliötä ovat erikokoisia, vertaa vain niitä elementtejä keskenään, joiden koot ovat yhteisiä molemmille. Heti, kun elementti ei ole yhtäsuuri, palauta arvo false. Jos kaikki verratut elementit ovat yhtäsuuria, palauta tietenkin true. Käy vektori läpi käyttäen iteraattoria — käytä aiempaa esimerkkiä mallina. Kirjoita main()-funktio, jotta voisit kokeilla is_equal()-funktiota.

3.11 Kompleksilukujen tyypit

Kompleksilukuluokka on osa vakiokirjastoa. Jotta sitä voitaisiin käyttää, pitää ottaa mukaan siihen liittyvä otsikkotiedosto:

```
#include <complex>
```

Kompleksiluvulla on kaksi osaa: reaalilukuosa ja imaginäärinen lukuosa. Imaginäärinen luku edustaa negatiivisen luvun neliöjuurta. Termin loi alun perin Descartes. Kompleksiluvun yleinen merkintätapa on

```
2 + 3i
```

jossa 2 edustaa reaaliosaa ja 3i edustaa imaginääristä osaa. Nämä kaksi osaa edustavat yksittäistä lukua.

Kompleksiolion määrittely tapahtuu jollakin seuraavista yleisistä muodoista:

```
// aito imaginäärinen luku: 0 + 7i
complex< double > purei( 0, 7 );

// imaginäärisen osan oletusarvo on nolla: 3 + 0i
complex< float > real_num( 3 );

// reaali- ja imaginäärisen osan oletusarvo on nolla: 0 + 0i
complex< long double > zero;

// alusta kompleksiolio toisella:
complex< double > purei2( purei );
```

Tässä kompleksioliota edustivat float, double tai long double. Voidaan esitellä myös kompleksilukuolioiden taulukko:

```
complex< double > conjugate[ 2 ] = {
                        complex< double >( 2, 3 ),
                        complex< double >( 2, -3 )
};
```

Voimme esitellä myös osoittimen tai viittauksen:

```
complex< double > *ptr = &conjugate[0];
complex< double > &ref = *ptr;
```

Kompleksiluvut tukevat lisäystä, vähennystä, kertomista, jakamista ja yhtäsuuruuden testaamista. Lisäksi on olemassa tuki sekä reaali- että imaginääriosien käsittelylle. Näitä operaatioita katsotaan yksityiskohtaisesti kohdassa 4.6.

3.12 Typedef-nimet

Typedef-mekanismi tarjoaa yleisen keinon esitellä mnemonisia synonyymejä sisäisille tai käyttäjän määrittelemille tietotyypeille. Esimerkiksi:

```
typedef double      wages;
typedef vector<int> vec_int;
typedef vec_int     test_scores;
typedef bool        in_attendance;
typedef int         *Pint;
```

Nämä typedef-nimet voivat toimia tyyppimääritteinä ohjelmassa:

```
// double hourly, weekly;
wages hourly, weekly;

// vector<int> vec1( 10 );
vec_int vec1( 10 );

// vector<int> test0( class_size );
const int class_size = 34;
test_scores test0( class_size );

// vector< bool > attendance;
vector< in_attendance > attendance( class_size );

// int *table[ 10 ];
Pint table[ 10 ];
```

Typedef-määrittely alkaa avainsanalla typedef, jonka jälkeen tulevat tietotyyppi ja tunnus. Tunnus eli typedef-nimi ei esittele uutta tyyppiä, vaan sen sijaan synonyymin olemassaolevalle tietotyypille. Typedef-nimi voi esiintyä ohjelmassa missä tahansa, missä tyyppinimikin voi esiintyä.

Typedef-nimi voi toimia ohjelman dokumentaation apuna. Se voi myös vähentää esittelyn merkintätavan monimutkaisuutta. Typedef-nimiä käytetään tyypillisesti esimerkiksi parantamaan monimutkaisten malliesittelyiden määrittelyiden luettavuutta (katso kohdasta 3.14 esimerkki), osoittimiin funktioihin (käsitellään kohdassa 7.9) ja osoittimiin luokan jäsenfunktioihin (käsitellään kohdassa 13.6).

Tässä on kysymys, johon melkein kaikki vastaavat väärin ensimmäisellä kerralla. Virhe on

käsittää typedef kirjaimellisesti makrolaajennuksena. Kun on annettu seuraava typedef:

```
typedef char *cstring;
```

mikä on cstr:n tyyppi seuraavassa esittelyssä?

```
extern const cstring cstr;
```

Vastaus on melkein aina

```
const char *cstr
```

Tämä tarkoittaa osoitinta vakiomerkkiin. Mutta se on väärin. const muokkaa cstr:n tyyppiä. cstr on osoitin ja siitä syystä määrittely esittelee cstr:n const-osoittimeksi char:iin (katso kohdasta 3.5 const-osoitintyyppien käsittely):

```
char *const cstr;
```

3.13 Volatile-määre

Olio esitellään volatile-määreellä, kun sen arvoa voidaan mahdollisesti muuttaa tavoilla, jotka ovat kääntäjän kontrollin tai havaitsemisen ulkopuolella — esimerkiksi muuttuja, jota järjestelmän kello ylläpitää. Tiettyjä optimointeja, joita kääntäjä tavallisesti tekee, ei tulisi käyttää olioihin, jotka ohjelmoija varustaa volatile-määreellä.

Määrettä volatile käytetään paljon samalla tavalla kuin const-määrettä — tyypin lisämääreenä. Esimerkiksi:

```
volatile int display_register;
volatile Task *curr_task;
volatile int ixa[ max_size ];
volatile Screen bitmap_buf;
```

display_register on int-tyyppinen volatile-olio. curr_task on osoitin volatile Task -luokkaolioon. ixa on volatile-kokonaislukujen taulukko. Taulukon jokaista elementtiä pidetään volatile-tyyppisenä. bitmap_buf on volatile Screen -luokkaolio. Jokaista sen tietojäsentä pidetään volatile-tyyppisenä.

volatile-määreen varsinainen käyttötarkoitus on ilmoittaa kääntäjälle, että olio voi muuttua tavoilla, joita kääntäjä ei havaitse. Siitä syystä kääntäjän ei tulisi voimakkaasti optimoida koodia, joka viittaa olioon.

3.14 Parityyppi

Pariluokka (pair), joka on osa vakiokirjastoa, mahdollistaa kahden joko samaa tai eri tyyppiä olevan arvon liittämisen yksittäiseen olioon. Jos pariluokkaa halutaan käyttää, pitää ottaa mukaan seuraava otsikkotiedosto:

```
#include <utility>
```

Esimerkiksi

```
pair< string, string > author( "James", "Joyce" );
```

luo olioparin author, joka muodostuu kahdesta merkkijono-oliosta, ja se alustetaan arvoilla "James" ja "Joyce".

Parin yksittäisiä elementtejä voidaan käsitellä *jäsenen käsittelyn merkitsemistavalla*: first (ensimmäinen) ja second (toinen). Esimerkiksi:

```
string firstBook;

if ( joyce.first == "James" &&
    joyce.second == "Joyce" )
    firstBook = "Stephen Hero";
```

Jos haluamme määritellä monia samantyyppisiä olioita, on mukavinta käyttää typedef-määrettä kuten seuraavassa:

```
typedef pair< string, string > Authors;

Authors proust( "marcel", "proust" );
Authors joyce( "james", "joyce" );
Authors musil( "robert", "musil" );
```

Tässä on toinen pari. Ensimmäinen sisältää olion nimen ja toinen sisältää osoittimen symbolitaulukkoon:

```
// alkuesittelyt
class EntrySlot;
extern EntrySlot* look_up( string );

typedef pair< string, EntrySlot* > SymbolEntry;

SymbolEntry current_entry( "author", look_up( "author" ));
// ...

if ( EntrySlot *it = look_up( "editor" ))
{
    current_entry.first = "editor";
    current_entry.second = it;
}
```

Kohtaamme parityypin jälleen, kun käsittelemme vakiokirjaston säiliötyyppejä luvussa 6 ja geneerisiä algoritmeja luvussa 12.

3.15 Luokkatyypit

Luokkamekanismi tukee uusien luokkatyyppien, kuten oliopohjaisen merkkijonon, vektorin, kompleksiluvun ja parin suunnittelua, joita tässä luvussa on käsitelty yhtä hyvin kuin luvussa 1 esiteltyä oliokeskeistä iostream-luokkahierarkiaa. Luvussa 2 kävimme läpi taustaan liittyvät käsitteet ja mekanismit, jotka tukevat oliopohjaisten ja oliokeskeisten luokkien suunnittelua toteuttamalla ja kehittämällä Array-luokan abstraktiota. Tässä kohtaa käymme lyhyesti läpi

yksinkertaisen oliopohjaisen String-luokka-abstraktion suunnittelun ja toteutuksen. Tämän tulisi olla suhteellisen kiinnostavaa, koska käsittelimme aikaisemmin sekä C-tyylisen merkkijonon että vakiokirjaston string-tyypin. Erityisesti toteutuksemme tuo esille C++:n tuen *operaattoreiden ylikuormitukselle*, joka esiteltiin lyhyesti kohdassa 2.3. (Luokat esitellään yksityiskohtaisesti luvuissa 13 — 15. Kun tuomme esille joitakin luokkien piirteitä kirjan aikaisemmassa vaiheessa, se antaa mahdollisuuden esittää kiinnostavampia luokkia käyttäviä esimerkkejä ennen kuin pääsemme lukuun 13. Tätä ensi kertaa lukeva voi haluta hypätä tämän osan yli ja odottaa luokkien täydellisempää esittelyä myöhemmissä kappaleissa.)

Meillä tulisi olla tässä vaiheessa jo hyvä käsitys siitä, mitä String-luokkamme tulisi tehdä: tarvitsemme tukea String-oliolle sekä alustamiseen että sijoittamiseen joko merkkijonoliteraalilla, C-tyylisellä merkkijonolla tai toisella String-oliolla. Toteutamme tämän erityisellä luokan alustusfunktiolla ja luokkakohtaisilla sijoitusoperaattorin ilmentymillä.

Meidän tulee tukea indeksointia yksittäisten String-olion merkkien käsittelyssä samaan tapaan kuin C-tyylisessä merkkijonossa ja vakiokirjaston string-tyypissä. Toteutamme tämän luokkakohtaisella indeksioperaattorin ilmentymällä.

Lisäksi haluaisimme tukea operaatioita, jotka päättelevät String-olion koon (`size()`), vertaa kahden String-olion tai String-olion ja C-tyylisen merkkijonon yhtäsuuruutta ja String-olion lukua ja kirjoitusta. Toteutamme nämä kaksi viimeistä operaatiota luokkakohtaisilla yhtäsuuruus-, iostream syöttö- ja iostream-tulostusoperaattorien ilmentymillä. Lopuksi meidän pitää päästä taustalla olevaan C-tyyliseen merkkijonoon.

Luokan määrittely muodostuu avainsanasta `class` ja tunnuksesta, joka toimii luokan tyyppimääritteenä kuten complex, vector, Array jne. Yleensä luokka muodostuu operaatioiden `public`-osasta ja tiedon `private`-osasta. Näitä operaatioita kutsutaan vaihtelevasti joko luokan *jäsenfunktioiksi* tai *metodeiksi*. Ne määrittelevät luokan *julkisen rajapinnan* — joukon operaatioita, joita käyttäjä voi tehdä luokan olioilla. String-luokkamme yksityinen tieto muodostuu seuraavista: `_string`, joka on tyyppiä `char*` ja osoittaa dynaamisesti varattuun merkkijonoon ja `_size`, joka on tyyppiä `int` ja sisältää String-olion koon. Tässä on luokkamäärittelymme:

```
#include <iostream>

class String;
istream& operator>>( istream&, String& );
ostream& operator<<( ostream&, const String& );

class String {
public:
        // ylikuormitettujen muodostajien joukko
        // tarjoaa automaattista alustusta
        // String str1;             // String()
        // String str2( "literal" ); // String( const  char* );
        // String str3( str2 );      // String( const String& );

        String();
        String( const char* );
        String( const String& );

        // tuhoaja: automaattinen tuhoaminen
        ~String();

        // ylikuormitettujen sijoitusoperaattorien joukko
        // str1 = str2
        // str3 = "a string literal"

        String& operator=( const String& );
        String& operator=( const char* );

        // ylikuormitettujen yhtäsuuruusoperaattoreiden joukko
        // str1 == str2;
        // str3 == “a string literal”;

        bool operator==( const String& );
        bool operator==( const char* );

        // ylikuormitettu indeksioperaattori
        // str1[ 0 ] = str2[ 0 ];

        char& operator[]( int );

        // jäsenten käsittelyfunktiot
        int   size()  { return _size;   }
        char* c_str() { return _string; }

private:
        int   _size;
        char *_string;

};
```

String-luokkaamme määritellään kolme muodostajaa. Kuten käsittelimme lyhyesti kohdassa 2.3, funktion ylikuormitusmekanisimi sallii samannimisen funktion tai operaattorin käytön useissa ilmentymissä edellyttäen, että jokainen niistä voidaan erottaa toisistaan parametriluettelonsa perusteella. Kolmen muodostajamme joukko on kelvollinen ylikuormitetuiksi funktioiksi ensiksikin parametrien määrän perusteella ja toiseksi tyypin perusteella. Ensimmäistä

```
String();
```

kutsutaan *oletusmuodostajaksi*, koska se ei vaadi eksplisiittistä alkuarvoa. Kun kirjoitamme

```
String str1;
```

käytetään oletusmuodostajaa str1:lle.

Kummallekin kahdelle muulle String-muodostajalle annetaan yksi argumentti. Kun kirjoitamme

```
String str2( "merkkijonoliteraali" );
```

käytetään muodostajaa

```
String( const char* );
```

str2:lle argumentin tyyppiin perustuen. Samalla tavalla, kun kirjoitamme

```
String str3( str2 );
```

käytetään muodostajaa

```
String( const String& );
```

str3:lle — jälleen tämä päätellään muodostajalle välitetyn argumentin tyypin perusteella. Tätä muodostajaa kutsutaan *kopiointimuodostajaksi*, koska se alustaa yhden luokan olion toisen kopiolla. Kun kirjoitamme

```
String str4( 1024 );
```

ei todellisen argumentin tyyppi vastaa yhtäkään parametrityyppiä, jota muodostajajoukko edellyttää, joten str4:n määrittely johtaa käännöksenaikaiseen virheeseen.

Ylikuormitetulle operaattorille käytetään yleistä muotoa

```
paluu_tyyppi operator op ( parametri_luettelo );
```

jossa operator on avainsana ja op on jokin esimääritellyistä operaattoreista kuten +, =, ==, [] jne. (täsmälliset säännöt käsitellään luvussa 15). Seuraavassa

```
char& operator[]( int );
```

esitellään ylikuormitettu ilmentymä indeksioperaattorista, joka saa yhden int-tyyppisen argumentin ja palauttaa viittauksen char-tyyppiin. Itse ylikuormitettu operaattori voidaan ylikuormittaa edellyttäen, että erillisten ilmentymien parametriluettelot voidaan erottaa toisistaan. Luomme esimerkiksi kaksi erilaista esiintymää sijoitus- ja yhtäsuuruusoperaattoreista String-luokallemme.

Nimetty jäsenfunktio käynnistetään käyttäen jäsenen käsittelyn merkintätapaa. Jos esimerkiksi on annettu seuraavat String-määrittelyt:

```
String object( "Danny" );
String *ptr = new String( "Anna" );
String array[2];
```

jäsenfunktio size() voidaan käynnistää kuten seuraavassa ja palauttaa vastaavasti arvot 5, 4 ja 0 (katsomme String-luokkamme toteutusta hetken kuluttua).

```
vector<int> sizes( 3 );

// jäsenen käsittelyn merkintätapa pisteellä olioille;
// olion koko on 5
sizes[ 0 ] = object.size();

// jäsenen käsittelyn merkintätapa nuolella osoittimille
// ptr:n koko on 4
sizes[ 1 ] = ptr->size();

// jäsenen käsittelyn merkintätapa pisteellä jälleen
// array[0]:n koko on 0
sizes[ 2 ] = array[0].size();
```

Ylikuormitettuja operaattoreita käytetään suoraan luokkaolioihin. Esimerkiksi:

```
String name1( "Yadie" );
String name2( "Yodie" );

// käyttää: bool operator==(const String&)
if ( name1 == name2 )
    return;
else
// käyttää: String& operator=( const String& )
   name1 = name2;
```

Luokan jäsenfunktiot voidaan määritellä joko luokkamäärittelyn sisällä tai ulkopuolella. (Esimerkiksi sekä size() että c_str() on määritelty String-luokkamäärittelyssämme.) Niistä jäsenfunktioista, jotka on määritelty luokkamäärittelyn ulkopuolella, pitää informoida kääntäjää, ei vain niiden nimillä, palautustyypillä ja parametriluettelolla, vaan myös tiedolla, mihin luokkaan ne kuuluvat. Jäsenfunktion määrittely tulisi sijoittaa ohjelman tekstitiedostoon — esimerkkinä String.C — ja sen pitää ottaa mukaan otsikkotiedosto, joka sisältää luokan määrittelyn — meidän esimerkissämme otsikkotiedoston String.h. Esimerkiksi:

```
// tämä sijoitetaan ohjelman tekstitiedostoon: String.C

// ota mukaan String-luokan määrittely
#include "String.h"

// ota mukaan strcmp()-funktion esittely
// cstring on C-standardin kirjaston otsikkotiedosto
#include <cstring>
```

```
bool                        // palautustyyppi
String::                    // luokka
operator==                  // funktion nimi: yhtäsuuruusoperaattori
(const String &rhs)// parametriluettelo
{
   if ( _size != rhs._size )
       return false;
   return strcmp( _string, rhs._string ) ? false : true;
}
```

strcmp() on, kuten muistat, C-standardin kirjastofunktio. Se vertailee kahta C-tyylistä merkkijonoa. Se palauttaa arvon 0, jos ne ovat yhtäsuuria; muussa tapauksessa muun kuin nollan. *Ehdollinen operaattori* (?:) testaa ehtoa ennen kysymysmerkkiä. Jos se on tosi, valitaan lauseke, joka on kysymysmerkin ja kaksoispisteen välissä; jos se on epätosi, valitaan lauseke, joka on kaksoispisteen jälkeen. Esimerkissämme ehdollinen operaattori palauttaa arvon false, jos strcmp() palauttaa muun kuin nolla-arvon; muussa tapauksessa se palauttaa arvon true. (Kohdassa 4.7 käsitellään ehdollista operaattoria yksityiskohtaisesti.)

Koska yhtäsuuruusoperaattori on pieni funktio, jota todennäköisesti kutsutaan säännöllisesti, on hyvä ajatus esitellä se välittömäksi (inline) funktioksi. Välittömän funktion lähdekoodi laajennetaan ohjelmaan jokaiseen funktion käynnistyspisteeseen tarkoituksena eliminoida funktiokutsujen kuormaa. Välittömällä funktiolla voidaan saavuttaa merkittävä suorituskyvyn parannus edellyttäen, että funktio käynnistetään riittävän monta kertaa. (Välittömät funktiot käsitellään yksityiskohtaisesti kohdassa 7.6.) Jäsenfunktiosta, joka on määritelty luokan määrittelyn yhteydessä kuten size(), tehdään välitön oletusarvoisesti. Jäsenfunktion, joka on määritelty luokan ulkopuolella, pitää esitellä itsensä eksplisiittisesti välittömäksi:

```
inline bool
String::operator==(const String &rhs)
{
        // kuten edellä
}
```

Välittömään jäsenfunktioon, joka on määritelty luokan rungon ulkopuolella, pitäisi ottaa mukaan otsikkotiedosto, joka sisältää luokan määrittelyn. Kun määrittelemme uudelleen yhtäsuuruusoperaattorin, pitäisi meidän siirtää sen määrittely lähdekoodista String.C otsikkotiedostoon String.h.

Tässä on yhtäsuuruusoperaattorimme, joka vertailee String-oliota C-tyyliseen merkkijonoon (se on myös määritelty välittömäksi ja sijoitettu String.h-otsikkotiedostoon).

```
inline bool
String::operator==(const char *s)
{
   return strcmp( _string, s ) ? false : true;
}
```

Muodostaja on saanut saman nimen kuin luokka. Sen ei pidä määrittää paluuarvoa esittelyssään taikka muodostajan rungossa. Ei yhtäkään tai kaikki esiintymistä voidaan esitellä välittömiksi.

```
#include <cstring>

// oletusmuodostaja
inline String::String()
{
    _size = 0;
    _string = 0;
}

inline String::String( const char *str )
{
     if ( ! str ) {
          _size = 0; _string = 0;
          }
          else {
                    _size = strlen( str );
                    _string = new char[ _size + 1 ];
                    strcpy( _string, str );
          }
}

// kopiointimuodostaja
inline String::String( const String &rhs )
{
    _size = rhs._size;
    if ( ! rhs._string )
        _string = 0;
    else {
        _string = new char[ _size + 1 ];
        strcpy( _string, rhs._string );
    }
}
```

Koska varaamme muistia dynaamisesti käyttämällä new-lauseketta merkkijonoa varten, pitää tuo muisti vapauttaa käyttäen delete-lauseketta, kun String-oliota ei enää tarvita. Tämä voidaan saavuttaa automaattisesti määrittelemällä luokan tuhoaja ja sijoittamalla delete-lauseke sen sisään. Jos luokan tuhoaja on määritelty, käynnistetään se automaattisesti jokaiselle luokkaoliolle sen elinajan päättymisen jälkeen. (Luvussa 8 kuvataan olion kolme mahdollista elinaikaa.) Tuhoaja yksilöidään antamalla sille luokan nimi ja laittamalla sen eteen tilde (~). Tässä on määrittelymme String-luokan tuhoajasta:

```
inline String::~String() { delete [] _string; }
```

Kaksi ylikuormitettua sijoitusoperaattoria viittaavat erityiseen avainsanaan nimeltään this. Kun kirjoitamme

```
String name1( "orville" ), name2( "wilbur" );
name1 = "Orville Wright";
```

this osoittaa name1:een sijoitusoperaattorissamme.

Yleisemmin sanottuna this-osoitin asetetaan automaattisesti osoittamaan jäsenfunktiossa luokan vasemmanpuoleista oliota, jonka kautta jäsenfunktio on käynnistetty. Kun kirjoitamme

```
ptr->size();
obj[ 1024 ];
```

niin size()-jäsenfunktiossa this-osoitin osoittaa ptr:ää; indeksioperaattorin kohdalla this-osoitin osoittaa obj:tä. Kun kirjoitamme *this, käsittelemme varsinaista oliota, jota this osoittaa (kohdassa 13.4 käsitellään this-osoitin yksityiskohtaisesti).

```
inline String&
String::operator=( const char *s )
{
   if ( ! s ) {
            _size = 0;
            delete [] _string;
            _string = 0;
   }
   else {
            _size = strlen( s );
            delete [] _string;
            _string = new char[ _size + 1 ];
            strcpy( _string, s );
   }
   return *this;
}
```

Eräs yleinen sudenkuoppa on, kun kopioidaan yksi luokka toiseen, että unohdetaan ensiksi testata, ovatko nuo kaksi luokkaoliota todella samanlaisia. Tämä virhe tapahtuu tyypillisimmin silloin, kun molempiin olioihin viitataan osoittimella käänteisesti. Silloin tulee this-osoitin jälleen avuksi tukemaan testiä. Esimerkiksi:

```
inline String&
String::operator=( const String &rhs )
{
   // lausekkeessa
   // name1 = *pointer_to_string
   // this osoittaa name1:een,
   // rhs edustaa *pointer_to_string.
   if ( this != &rhs ) {
```

Tässä on koko toteutus:

```
inline String&
String::operator=( const String &rhs )
{
        if ( this != &rhs )
        {
           delete [] _string;
           _size = rhs._size;

           if ( ! rhs._string )
                        _string = 0;
           else {
                        _string = new char[ _size + 1 ];
                        strcpy( _string, rhs._string );
           }
        }
        return *this;
}
```

Indeksioperaattori on lähes identtinen sen kanssa, jonka toteutimme Array-luokallemme kohdassa 2.3:

```
#include <cassert>

inline char&
String::operator[]( int elem )
{
   assert( elem >= 0 && elem < _size );
   return _string[ elem ];
}
```

Syöttö- ja tulostusoperaattorit on toteutettu muina kuin jäsenfunktioina. (Syytä tähän käsitellään kohdassa 15.2. Kohdissa 20.4 ja 20.5 on yksityiskohtainen selvitys iostream-kirjaston syöttö- ja tulostusoperaattoreiden ylikuormituksesta.) Meidän syöttöoperaattorimme lukee enintään 4095 merkkiä. setw() on esimääritelty iostream-kirjaston tulostuksen muotoilija (manipulaattori). Se lukee enintään yhden vähemmän kuin arvo, joka sille välitetään, ja siten takaa, että inBuf-merkkitaulukkomme ei vuoda yli. Jotta voisimme käyttää sitä, meidän pitää ottaa mukaan iomanip-otsikkotiedosto. (Luvussa 20 käsitellään setw() tarkemmin.)

```
#include <iomanip>

inline istream&
operator>>( istream &io, String &s )
{
   // keinotekoinen rajoitus 4096 merkin lukemiseen
   const int limit_string_size = 4096;
   char inBuf[ limit_string_size ];
```

```
    // setw() on osa iostream-kirjastoa
    // rajoittaa luettavien merkkien määrän: limit_string_size-1
    io >> setw( limit_string_size ) >> inBuf;

    s = inBuf; // String::operator=( const char* );
    return io;
}
```

Tulostusoperaattori tarvitsee pääsyn taustalla olevaan char*-esitystapaan, jotta String voitaisiin näyttää. Koska se ei kuitenkaan ole luokan jäsenfunktio, sillä ei ole pääsyoikeuksia _string:iin. On olemassa kaksi mahdollista ratkaisua: ensimmäinen on myöntää erityisoikeudet tulostusoperaattorille (tämä tehdään esittelemällä se luokan *ystäväksi* (*friend*) — katsomme tätä kohdassa 15.2). Toinen ratkaisu on tehdä välitön käsittelyfunktio — tässä tapauksessa c_str(), jonka malli on saatu vakiokirjaston string-luokan ratkaisusta. Tässä on toteutuksemme:

```
inline ostream&
operator<<( ostream& os, const String &s )
{
    return os << s.c_str();
}
```

Seuraava pieni ohjelma testaa String-luokkamme toteutusta. Se lukee merkkijonoja (String-olioita) vakiosyötöstä ja käy ne läpi jokaisen vuorollaan pitäen kirjaa löydetyistä vokaaleista.

```
#include <iostream>
#include "String.h"
int main()
{
    int aCnt = 0, eCnt = 0, iCnt = 0, oCnt = 0, uCnt = 0,
        theCnt = 0, itCnt = 0, wdCnt = 0, notVowel = 0;

    // emme määrittele The( "The" ) eikä It( "It" )
    // käyttääksemme operaattoria operator==( const char* )
    String buf, the( "the" ), it( "it" );

    // käynnistää operaattorin operator>>( ostream&, String& )
    while ( cin >> buf ) {
        ++wdCnt;

        // käynnistää operaattorin operator<<( ostream&, const String& )
        cout << buf << ' ';

        if ( wdCnt % 12 == 0 )
            cout << endl;

        // käynnistää operaattorit String::operator==(const String&) ja
        //         String::operator==( const char* );
        if ( buf == the || buf == "The" )
```

```
            ++theCnt;
        else
        if ( buf == it || buf == "It" )
            ++itCnt;

        // käynnistää: String::size()
        for ( int ix = 0; ix < buf.size(); ++ix )
        {
            // käynnistää operaattorin String::operator[](int)
            switch( buf[ ix ] )
            {
                case 'a': case 'A': ++aCnt; break;
                case 'e': case 'E': ++eCnt; break;
                case 'i': case 'I': ++iCnt; break;
                case 'o': case 'O': ++oCnt; break;
                case 'u': case 'U': ++uCnt; break;
                default: ++notVowel; break;
            }
        }
    }

    // käynnistää operaattorin operator<<( ostream&, const String& )
    cout << "\n\n"
        << "Luetut sanat: " << wdCnt  << "\n\n"
        << "the/The: "    << theCnt << '\n'
        << "it/It: "      << itCnt  << "\n\n"
        << "muita kuin vokaaleita luettu: " << notVowel << "\n\n"
        << "a: " << aCnt << '\n'
        << "e: " << eCnt << '\n'
        << "i: " << iCnt << '\n'
        << "o: " << oCnt << '\n'
        << "u: " << uCnt << endl;
}
```

Ohjelman syöttöaineisto on kappale lastensadusta, jonka Stan on kirjoittanut (näemme sen jälleen luvussa 6). Kun ohjelma on käännetty ja toteutettu, ohjelmamme generoi seuraavan tulostuksen:

```
Alice Emma has long flowing red hair. Her Daddy says when the
wind blows through her hair, it looks almost alive, like a fiery
bird in flight. A beautiful fiery bird, he tells her, magical but
untamed. "Daddy, shush, there is no such thing," she tells him, at
the same time wanting him to tell her more. Shyly, she asks,
"I mean, Daddy, is there?"

Luetut sanat: 65

the/The: 2
it/It: 1
```

```
muita kuin vokaaleita luettu: 190

a: 22
e: 30
i: 24
o: 10
u: 7
```

Harjoitus 3.26

String-luokan muodostajien ja sijoitusoperaattoreiden toteutuksessa on paljon tuplakoodia. Yritä keskittää yhteinen koodi erilliseen yksityiseen jäsenfunktioon käyttäen apuna mallia, joka esiteltiin kohdassa 2.3. Toteuta muodostajat ja sijoitusoperaattorit uudelleen, jotta ne käyttäisivät sitä hyväkseen. Aja ohjelma uudelleen varmistuaksesi, että se toimii yhä.

Harjoitus 3.27

Muokkaa ohjelmaa niin, että se laskee myös konsonantit `b`, `d`, `f`, `s` ja `t`.

Harjoitus 3.28

Toteuta jäsenfunktio, joka laskee merkin esiintymät String-oliossa. Sen esittely on seuraavanlainen:

```
class String {
public:
   // ...
   int count( char ch ) const;
   // ...
};
```

Harjoitus 3.29

Toteuta jäsenfunktio operaattorille, joka yhdistää yhden String-olion toiseen ja palauttaa uuden String-olion. Sen esittely on seuraavanlainen:

```
class String {
public:
   // ...
   String operator+( const String &rhs ) const;
   // ...
};
```