4

Lausekkeet

Luvussa 3 tarkastelimme sisäisiä ja vakiokirjaston tukemia tietotyyppejä. Tässä luvussa katsomme esimääriteltyjä operaattoreita kuten lisäys, vähennys, sijoitus ja yhtäsuuruuden testaus jne., joita käytetään tiedon käsittelyyn. Kun se on tehty, tarkastelemme operaattoreiden sidontajärjestystä. Jos esimerkiksi on annettu lauseke `3+4*5`, on tulos aina 23 eikä 35, koska kertomisoperaattorilla on korkeampi sidontajärjestys ja myöhempi arviointijärjestys. Lopuksi katsomme aihetta olioiden sekä implisiittisistä että eksplisiittisistä konversioista. Esimerkiksi lausekkeessa `3+0.7` kokonaislukuarvo 3 aina ylennetään liukuluvuksi ennen kuin yhteenlasku suoritetaan.

4.1 Mikä on lauseke?

Lauseke muodostuu yhdestä tai useammasta *operandista*. Lausekkeen yksinkertaisin muoto muodostuu yhdestä literaalivakiosta tai oliosta. Tuloksena on yleensä operandi *rvalue*. Esimerkiksi tässä on kolme yksinkertaista lauseketta:

```
void mumble()
{
        3.14159;
        "melancholia";
        upperBound;
}
```

Literaalivakion `3.14159` tulos on `3.14159`. Sen tyyppi on `double`. Merkkijonon `"melancholia"` tulos on sen ensimmäisen elementin osoite muistissa. Sen tyyppi on `const char*`. Olion `upperBound` tulos on siihen liittyvä arvo. Sen tyyppi päätellään sen määrittelyn perusteella.

Yleisemmin sanottuna lauseke muodostuu yhdestä tai useammasta operandista ja operaatiosta, jota niihin käytetään. Esimerkiksi kaikki seuraavat ovat lausekkeita (olemme jättäneet pois olioiden määrittelyt; käytetään automaattisesti sopivaa operaatiota, joka perustuu operandin (tai operandien) tyyppiin):

```
salary + raise
ivec[ size/2 ] * delta
```

```
first_name + " " + last_name
```

Operaattorit edustavat operaatioita, joita operandeihin kohdistetaan. Esimerkiksi ensimmäisessä lausekkeessa käytetään liukuluvun lisäysoperaattoria operandeihin salary ja raise. Toisessa lausekkeessa operandi size jaetaan luvulla 2. Tulosta käytetään kokonaislukutaulukon ivec indeksointiin. Sen arvo kerrotaan operandilla delta. Kolmannessa lausekkeessa yhdistetään kaksi merkkijono-operandia ja merkkijonoliteraali niin, että ne muodostavat uuden merkkijonon käyttäen yhteenlaskuoperaattoria, joka on määritelty vakiokirjaston string-luokassa.

Operaattorit, jotka vaikuttavat vain yhteen operandiin, ovat *unaarisia* operaattoreita kuten osoite- (&) ja käänteisviittausoperaattori (*), kun taas operaattorit, jotka vaikuttavat kahteen operandiin, kuten lisäys- ja vähennysoperaattorit, ovat *binäärisiä* operaattoreita. Jotkut operaattorit voivat edustaa sekä unaarista että binääristä operaatiota (tarkemmin sanottuna samaa symbolia käytetään kahden eri operaattorin esittämiseen). Esimerkiksi

```
*ptr
```

edustaa unaarista käänteisviittausoperaattoria. Se palauttaa arvon, joka on tallennettu olioon, johon ptr osoittaa. Kuitenkin

```
var1 * var2
```

edustaa binääristä kertomisoperaattoria. Se laskee kahden operandin, var1 ja var2, tulon.

Lausekkeen arvioinnissa suoritetaan yksi tai useampi operaatio, joka johtaa tulokseen. Lausekkeen tulos on rvalue, paitsi jos on toisin mainittu. Aritmeettisen lausekkeen tuloksen tietotyyppi päätellään operandin (tai operandien) tyypin perusteella. Kun mukana on useampi kuin yksi tietotyyppi, tapahtuu tyyppikonversio, joka noudattaa esimääriteltyjä tyyppikonversiosääntöjä. (Katsomme tyyppikonversioita yksityiskohtaisesti kohdassa 4.14.)

Kun kaksi tai useampia operaattoreita yhdistetään, kutsutaan lauseketta *yhdistetyksi lausekkeeksi* (*compound expression*). Esimerkiksi seuraavan lausekkeen tarkoitus on päätellä, osoittaako osoitin ptr oliota (se osoittaa oliota, jos sen arvo ei ole nolla) ja onko sen osoittaman olion arvo muu kuin nolla[1]:

```
ptr != 0 && *ptr != 0
```

Koko lauseke muodostuu kolmesta alilausekkeesta: ptr:n erisuuruuden vertailusta nollaa vastaan, ptr:n käänteisviittauksesta ja siitä tuloksena syntyvän arvon testauksesta nollaa vastaan. Jos ptr on määritelty kuten seuraavassa

```
int ival = 1024;
int *ptr = &ival;
```

1. Eksplisiittinen testaus nollaa vastaan on valinnainen. Seuraava lauseke on yhtäpitävä ja sitä pidetään reaalimaailman C++-ohjelmien idiomina: ptr && *ptr

käänteisviittauksen alilausekkeen tulos on 1024 ja kahden alilausekkeen tulokset erisuuruuden vertailussa nollaa vastaan ovat molemmat tosia (true). Koko lausekkeen tulos on myös tosi: ptr:ää ei ole asetettu nollaksi ja oliota, jota se osoittaa, ei ole asetettu nollaksi. (&&-operaattoria kutsutaan JA-operaattoriksi (*AND*): se arvioidaan todeksi, jos sen sekä vasemman- että oikeanpuoleinen alilauseke arvioidaan todeksi; muussa tapauksessa se arvioidaan epätodeksi.)

Jos katsomme yhdistettyä lausekettamme lähemmin, huomaamme, että sen onnistunut arviointi riippuu alilausekkeiden arviointijärjestyksestä. Jos esimerkiksi lausekkeen toinen puoli arvioidaan ensiksi — eli jos ptr:ään viitataan käänteisesti ennen kuin on varmistuttu, että se ei ole nolla — ohjelman suoritus todennäköisesti epäonnistuu tai korruptoituu suorituksen aikana aina, kun ptr on asetettu arvoon 0. Kun JA-operaattori on mukana, alilausekkeiden arviointijärjestys on tarkkaan määritelty: jos vasen alilauseke arvioidaan epätodeksi, ei oikeanpuoleista alilauseketta arvioida, eikä silloin ohjelmavirhettä tapahdu.

Alilausekkeiden arviointijärjestys on käytännössä säännöllinen ohjelmavirheen lähde aloitteleville C- ja C++-ohjelmoijille. Sitä on vaikea huomata, koska ohjelmamme visuaalinen tutkinta ei tuo virhettä esille, ellemme ymmärrä alilausekkeiden arviointisääntöjä. Yleensä alilausekkeen arviointijärjestys päätellään sen operaattoreiden *sidontajärjestyksestä* (*precedence*) ja *assosiatiivisuudesta* (*associativity*). Katsomme tätä jollakin tarkkuudella kohdassa 4.13 sen jälkeen, kun katsomme C++:n tukemia operaattoreita. Seuraavissa kohdissa käsitellään esimääriteltyjä C++-operaattoreita niiden tuttuusjärjestyksessä.

4.2 Aritmeettiset operaattorit

Taulukko 4.1Aritmeettiset operaattorit.

Operaattori	Toiminto	Käyttö
*	kertolasku	lauseke * lauseke
/	jakolasku	lauseke / lauseke
%	jakojäännös	lauseke % lauseke
+	yhteenlasku	lauseke + lauseke
-	vähennyslasku	lauseke - lauseke

Kokonaislukujen välinen jakaminen tuottaa kokonaisluvun. Jos osamäärä sisältää murto-osan, se katkaistaan. Esimerkiksi lausekkeiden

```
int ival1 = 21 / 6;
int ival2 = 21 / 7;
```

tuloksena on, että sekä ival1 että ival2 saa alkuarvokseen 3.

%-operaattori laskee kahden arvon välisen jakojäännöksen; ensimmäinen arvo jaetaan toisella. Tätä operaattoria voidaan käyttää vain operandeihin, jotka ovat kokonaistyyppejä (char, short, int ja long). Kun molemmat operandit ovat positiivisia, on tulos positiivinen. Jos kuitenkin

jompikumpi (tai molemmat) operandeista on negatiivinen, jakojäännöksen etumerkki on koneriippuvainen; siten siirrettävyyttä ei taata. %-operattoria kutsutaan toisella nimellä *moduluksek-si* tai *jakojäännös*operaattoriksi.

```
3.14 % 3; // käännöksenaikainen virhe: liukulukuoperandi
21 % 6;   // ok: tulos on 3
21 % 7;   // ok: tulos on 0
21 % -5;  // koneriippuvainen: tulos on -1 tai 1

int ival = 1024;
double dval = 3.14159;

ival % 12;   // ok: palauttaa arvon väliltä 0 — 11
ival % dval; // käännöksenaikainen virhe: liukulukuoperandi
```

Tietyissä tilanteissa aritmeettisen lausekkeen arviointi johtaa väärään tai tuntemattomaan arvoon. Näitä tilanteita kutsutaan *aritmeettisiksi poikkeuksiksi* (mutta ne eivät johda todellisen poikkeuksen heittämiseen). Aritmeettinen poikkeus voi johtua matematiikan luonteesta — kuten nollalla jako — tai tietokoneen luonteesta — kuten *ylivuoto* (jossa arvo on suurempi kuin oliotyypin koko, johon se sijoitetaan). Esimerkiksi 8-bittinen char voi sisältää enintään arvon 127 tai 255 riippuen siitä, onko se etumerkillinen vai ei. Seuraava kertolasku sijoittaa char-tyyppiin arvon 256, joka johtaa ylivuotoon:

```
#include <iostream>

int main() {
    char byte_value = 32;
    int ival = 8;

    // tavuarvolle käytettävissä olevan muistialueen ylivuoto
    byte_value = ival * byte_value;

    cout << "byte_value: " << static_cast<int>(byte_value) << endl;
}
```

Jotta arvo 256 voitaisiin esittää, tarvitaan 9 bittiä. Arvon 256 sijoitus byte_value-olioon johtaa ylivuotoon siihen liittyvällä muistialueella. Todellinen arvo, jonka byte_value sisältää, on tuntematon ja aiheuttaa todennäköisesti ongelmia suorituksen aikana. Esimerkiksi SGI-työasemalla byte_value asetetaan arvoon 0. Kun aluksi yritimme tulostaa sen käyttäen lauseketta

```
cout << "byte_value: " << byte_value << endl;
```

ohjelman tulostus näytti tältä:

```
byte_value:
```

Muutaman hetken pähkäilyn jälkeen tajusimme, että arvo 0 ASCII-merkistössä edustaa null-merkkiä ja tulostuu kirjaimellisesti aivan kuin sitä ei olisikaan. Kummallista lauseketta

```
static_cast<int>(byte_value)
```

kutsutaan *eksplisiittiseksi tyyppikonversioksi* (*cast*). Tyyppikonversio ohjaa kääntäjää konvertoimaan olion (tai lausekkeen) nykyisestä tyypistään ohjelmoijan määrittämäksi tyypiksi. Meidän tapauksessamme konvertoimme byte_value-olion int-tyyppiseksi olioksi. Nyt ohjelma tulostaa

```
byte_value: 0
```

Meidän tapauksessamme se, mikä on muuttunut, ei ole byte_value-olioon liittyvä arvo, vaan se, kuinka tulostusoperaattori tulkitsee sen. Kun tulostamme byte_value-olion, sen tulostuksen muodon päättää siihen liittyvä tyyppi. Kun sitä kohdellaan char-oliona, sen arvo yhdistetään siihen liittyvään ASCII-esitystapaan (esimerkiksi 12 edustaa rivinvaihtomerkkiä, 97 edustaa pientä *a*-kirjainta, 0 edustaa null-merkkiä jne); Nimenomaan tämä esitystapa tulostetaan, eikä arvo. Kun sitä toisaalta kohdellaan int-tyyppinä, sen arvo tulostetaan suoraan. (Tyyppikonversioita käsitellään kohdassa 4.14.)

Kun käsittelimme tyyppikonversiota ja epäonnistuimme yrityksessämme tulostaa byte_value-olio odotetulla tavalla, kuvauksemme keskeytyksestä tuo tietyssä mielessä esille sen, kuinka ohjelmamme käyttäytyvät odottamattomalla tavalla, ja meidän on pakko laittaa syrjään ohjelmointitehtävämme selvittääksemme, mikä on mennyt pieleen. Näennäisesti epäselvät tai epäkiinnostavat kielen piirteet kuten tietotyyppien koot vaikuttavat joskus käytännössä kirjoittamiimme ohjelmiin. Kieli ei saa siepattua ylivuotoa kuten byte_value-oliolle tapahtui, koska se edellyttäisi suorituksenaikaisen tarkistuksen jokaiselle laskutapahtumalle ja se on epäkäytännöllistä suorituskyvyn kannalta. Meidän pitää vain yksinkertaisesti olla valppaana tuolle mahdollisuudelle.

C++-standardin limits-otsikkotiedostossa on tietoja sisäisten tyyppien esitystavoista kuten minimi- ja maksimiarvo, jonka tyyppi voi sisältää. Lisäksi C-standardin climits-otsikkotiedostossa, joka on käytettävissä myös C++-käännösjärjestelmässä, on määriteltynä esikääntäjän makroja, joissa on samoja tietoja. Jos haluat nähdä, kuinka näitä otsikkotiedostoja käytetään ylitai alivuodon välttämiseksi, katso luvut 4 ja 5 julkaisusta [PLAUGER92].

Liukulukuaritmetiikka edustaa tarkkuuden lisäongelmaa: tietokoneessa on käytettävissä vain kiinteä määrä numeroita arvojen esittämiseen. Liukuluvun *pyöristyminen* tapahtuu, kun arvo muokataan sopimaan float-, double- tai long double -tyyppiin esitystapaa varten. Yhteen-, kerto- ja vähennyslaskun tuloksen tarkkuuteen vaikuttaa taustalla olevan tietotyypin kiinteä tarkkuus. (Jos haluat nähdä yksityiskohtaisen käsittelyn pyöristymisvirheestä numeerisessa laskennassa, katso julkaisusta [SHAMPINE97].)

Harjoitus 4.1

Mikä on pääasiallisin ero seuraavien jakolausekkeiden välillä?

```
double dval1 = 10.0, dval2 = 3.0;
int ival1 = 10, ival2 = 3;

dval1 / dval2;
ival1 / ival2;
```

Harjoitus 4.2

Kun on annettu kokonaisolio, mitä operaattoria voisimme käyttää päätelläksemme, onko se parillinen vai pariton? Kirjoita lauseke.

Harjoitus 4.3

Paikanna ja kokeile C++-standardin otsikkotiedostoa limits ja C-standardin otsikkotiedostoja climits ja cfloat järjestelmässäsi.

4.3 Yhtäsuuruus-, vertailu- ja loogiset operaattorit

Taulukko 4.2 Yhtäsuuruus-, vertailu- ja loogiset operaattorit.

Operaattori	Toiminto	Käyttö
!	looginen EI	!lauseke
<	pienempi kuin	lauseke < lauseke
<=	pienempi tai yhtäsuuri kuin	lauseke <= lauseke
>	suurempi kuin	lauseke > lauseke
>=	suurempi tai yhtäsuuri kuin	lauseke >= lauseke
==	yhtäsuuri	lauseke == lauseke
!=	erisuuri	lauseke != lauseke
&&	looginen JA	lauseke && lauseke
||	looginen TAI	lauseke || lauseke
Huomaa: näiden operaattorien tulos on bool-tyyppiä.		

Yhtäsuuruus-, vertailu- ja loogiset operaattorit arvioidaan bool-vakioiksi true tai false. Jos sellaista operaattoria käytetään yhteydessä, joka vaatii kokonaislukuarvoa, tulos ylennetään joko arvoon 1 (true) tai 0 (false). Esimerkiksi seuraavassa koodikatkelmassa haluamme laskea vektorin kaikki elementit, jotka ovat pienempiä kuin käyttäjän määrittelemä arvo. Tämän toteuttamiseksi lisäämme pienempi kuin -operaattorin tuloksen elementtien lukuun. (+=-ope-

raattori on lyhennetty merkintä ja tarkoittaa, että lisäämme oikeanpuoleisen lausekkeen arvon vasemmanpuoleisen lausekkeen nykyiseen arvoon. Käsittelemme yhdistettyjä sijoitusoperaattoreita kohdassa 4.4.)

```
int elem_cnt = 0;

vector<int>::iterator iter = ivec.begin();
while ( iter != ivec.end() )
{
    // sama kuin: elem_cnt = elem_cnt + (*iter < some_value)
    // true/false-arvo tästä: *iter < some_value
    //    ylennetään joko arvoon 1 tai 0
    elem_cnt += *iter < some_value;
    ++iter;
}
```

Looginen JA (&&) -operaattori arvioidaan true-arvoksi vain, jos sen molemmat operandit arvioidaan true-arvoiksi. Looginen TAI (||) -operaattori arvioidaan true-arvoksi, jos jompikumpi sen operandeista arvioidaan true-arvoksi. On taattu, että operandit arvioidaan vasemmalta oikealle. Arviointi loppuu heti, kun lausekkeen totuus tai epätotuus saadaan selville. Kun on annettu muodot

```
expr1 && expr2
expr1 || expr2
```

on taattu, että lauseketta expr2 ei arvioida, jos jokin seuraavista on totta:

- Loogisessa JA-lausekkeessa expr1 arvioidaan false-arvoksi.
- Loogisessa TAI-lausekkeessa expr1 arvioidaan true-arvoksi.

Arvokas käyttökohde loogiselle JA-operaattorille on tilanne, jossa annetaan lausekkeen expr1 tulla epätodeksi silloin, kun mukana on jokin raja-arvo, joka saisi lausekkeen expr2 arvioinnin vaaralliseksi. Esimerkiksi:

```
while ( ptr != 0 &&
                ptr->value < upperBound &&
                ptr->value >= 0 &&
                notFound( ia[ ptr->value ] ))
{ ... }
```

Osoitin, jonka arvo on 0, ei osoita oliota. Jäsenen käsittelyoperaattorin käyttö 0-arvoiseen osoittimeen on aina ongelmallista. Ensimmäinen looginen JA-operaattori estää sen mahdollisuuden. Samoin ongelmallinen on taulukon indeksi, joka on ylittänyt raja-arvot. Toinen ja kolmas operandi vartioivat, että sitä ei tapahtuisi. Viimeinen operandi on turvallinen arvioida, kun ensin kolme ensimmäistä operandia on arvioitu true-arvoiksi.

Looginen EI-operaattori (!) arvioidaan true-arvoksi, jos sen operandin arvo on false tai nolla; muussa tapauksessa se arvioidaan false-arvoksi. Esimerkiksi:

```
bool found = false;

// niin kauan, kun kohdetta ei ole löytynyt
// ja ptr yhä osoittaa oliota
while ( ! found && ptr ) {
   found = lookup( *ptr );
   ++ptr;
}
```

Alilauseke

```
! found
```

arvioidaan true-arvoksi niin kauan kuin found on yhtä kuin false. Se on lyhennetty merkintätapa eksplisiittiselle testille:

```
// !found merkitsee
found == false
```

Samalla tavalla testi

```
if ( found )
```

on lyhennetty merkintätapa eksplisiittiselle testille:

```
if ( found == true )
```

Vaikka binääristen vertailuoperaattoreiden kuten pienempi kuin- tai erisuuri kuin -operaattoreiden käyttö on selkeää, on olemassa eräs potentiaalinen kompastuskivi, josta meidän on syytä olla tietoisia: vasemmanpuoleisten ja oikeanpuoleisten operandien arviointijärjestys on jätetty tahallaan määrittelemättä sekä C- että C++-standardeissa, eikä se siten saa sisältää järjestysriippuvuuksia. Esimerkiksi seuraavassa

```
// hups! kieli ei määrittele arviointijärjestystä
if ( ia[ index++ ] < ia[ index ] )
   // vaihda elementit keskenään
```

ohjelmoija olettaa, että vasen operandi arvioidaan ensiksi ja että siksi ia[0] on vertailussa pienempi kuin ia[1]. Kieli ei kuitenkaan takaa arviointijärjestystä vasemmalta oikealle; toteutus voi itse asiassa arvoida oikeanpuoleisen operandin ensiksi, jolloin operandia ia[0] vertaillaan itseensä eikä operandia ia[1] koskaan arvioida. Eräs turvallinen ja siirrettävä uudelleentoteutus näyttää tältä:

```
if ( ia[ index ] < ia[ index+1 ] )
                // vaihda elementit keskenään
++index;
```

Toinen potentiaalinen kompastuskivi on seuraavassa. Aikomuksemme on päätellä, ovatko ival, jval ja kval yhtäsuuria arvoja. Näetkö, mikä on pielessä?

```
// hups! tämä ei päättele, ovatko 3 arvoa erisuuria
if ( ival != jval != kval )
   // tee jotain ...
```

Kuten toteutimme sen, arvot 0, 1 ja 0 saavat lausekkeen arvoksi true. Syynä tähän on se, että toisen erisuuruuslausekkeen vasen operandi on ensimmäisen operandin true/false-tulos — operandin kval erisuuruutta testataan ylennettyjä kokonaislukuarvoja 0 tai 1 vastaan. Saadaksemme testimme valmiiksi, voimme kirjoittaa lausekkeen uudelleen kuten tässä:

```
if ( ival != jval && ival != kval && jval != kval )
   // tee jotain ...
```

Harjoitus 4.4

Mitkä seuraavista ovat todennäköisesti virheellisiä tai eivät ole siirrettäviä, vai onko yksikään? Miksi? Miten ne voitaisiin korjata? (Huomaa, että olion (tai olioiden) tyyppi ei ole merkitsevässä asemassa näissä esimerkeissä.)

```
(a) ptr->ival != 0          (b) ival != jval < kval
(c) ptr != 0 && *ptr++      (d) ival++ && ival
(e) vec[ ival++ ] <= vec[ ival ];
```

Harjoitus 4.5

Binääristen operaattoreiden arviointijärjestys on jätetty tuntemattomaksi, mikä sallii kääntäjälle vapauden tehdä optimaalisen toteutuksen. Vaihtokauppaa käydään tehokkaan toteutuksen ja ohjelmoijan kielenkäytön potentiaalisten kompastuskivien välillä. Pidätkö sinä tätä hyväksyttävänä vaihtokauppana? Miksi tai miksi et?

4.4 Sijoitusoperaattorit

Alustus saa oliolle ensimmäisen arvon. Esimerkiksi:

```
int ival = 1024;
int *pi = 0;
```

Sijoitus puolestaan kirjoittaa olion nykyisen arvon päälle uuden arvon. Esimerkiksi:

```
ival = 2048;
pi = &ival;
```

Alustus ja sijoitus sekoitetaan joskus keskenään, koska ne molemmat käyttävät samaa operaattoria (=). Olio voidaan alustaa vain kerran: sen määrittelyhetkellä. Olioon voidaan sijoittaa monia kertoja ohjelman aikana.

Mitä tapahtuu, kun lauseke, johon sijoitetaan, ei ole samaa tyyppiä kuin siihen sijoitettavan olion tyyppi? Esimerkiksi:

```
ival = 3.14159; // ok?
```

Sääntö on, että oikeanpuoleisen lausekkeen tyypin pitää olla täsmälleen sen olion tyyppiä, johon se sijoitetaan. Tässä tapauksessa ival on tyyppiä int ja literaalivakio 3.14159 on tyyppiä double. Onko sijoitus virheellinen? Ei. Kääntäjä yrittää konvertoida implisiittisesti oikeanpuoleisen operandin tyypin vasemmanpuoleisen olion tyypiksi, johon se sijoitetaan. Jos tyyp-

pikonversio on mahdollista, kääntäjä toteuttaa sen hiljaisuudessa (jos tarkkuuden häviämistä tapahtuu kuten konversiossa tyypistä double tyyppiin int, annetaan yleensä varoitus). Esimerkissämme arvo 3.14159 konvertoidaan literaalivakioksi 3 ja tyypiksi int. Tämä on arvo, joka sijoitetaan operandiin ival.

Ellei implisiittinen tyyppikonversio ole mahdollinen, saa sijoitus aikaan käännöksenaikaisen virheen. Esimerkiksi seuraava sijoitus saa aikaan käännöksenaikaisen virheen, koska ei ole olemassa implisiittistä konversiota int -tyyppisestä arvosta int*-tyyppiseen arvoon:

```
pi = ival; // virhe
```

(Kielen tunnistamat implisiittiset tyyppikonversiot käsitellään kohdassa 4.14.)

Sijoitusoperaattorin vasemman operandin pitää olla *lvalue* — tarkoittaa, että siihen liittyy osoite, johon voidaan kirjoittaa. Selkeä esimerkki kelpaamattomasta ei-lvalue-sijoituksesta on seuraavassa:

```
1024 = ival; // virhe
```

Tässä on yksi mahdollinen ratkaisu:

```
int value = 1024;
value = ival; // ok
```

Joissakin tapauksissa lvalue ei itsessään ole kuitenkaan riittävä. Kun on tehty esimerkiksi seuraavat määrittelyt:

```
const int array_size = 8;
int ia[ array_size ] = { 0, 1, 2, 2, 3, 5, 8, 13 };
int *pia = ia;
```

on sijoitus

```
array_size = 512; // virhe
```

virheellinen, vaikka array_size on lvalue; array_size-olion const-määrittely tekee sen osoitteesta sellaisen, johon ei voi kirjoittaa. Samalla tavalla sijoitus

```
ia = pia; // virhe
```

ei ole kelvollinen. Vaikka ia on lvalue, ei itse taulukko-oliota voi sijoittaa siihen; vain sen sisältämiä elementtejä.

Sijoitus

```
pia + 2 = 1; // virhe
```

ei myöskään ole sallittu. Vaikka pia+2 johtaa osoitteeseen ia[2], ei tulos ole osoite, johon voi kirjoittaa. Jos kuitenkin osoitearvoon käytetään operaattoria käänteisesti kuten seuraavassa

```
*(pia + 2) = 1; // ok
```

on sijoitus ok: käänteisoperaattori ilmaisee, että sijoitus kohdistuu olioon, johon pia+2 viittaa.

Sijoituksen tulos on arvo, joka varsinaisesti sijoitetaan vasemman operandin muistipaikkaan. Esimerkiksi seuraavan tulos

```
ival = 0;
```

on 0, kun taas sijoituksen

```
ival = 3.14159;
```

tulos on 3; molemmat ovat int-tyyppisiä. Tästä syystä sijoitus voi olla alilauseke. Esimerkiksi seuraava while-silmukka

```
extern char next_char();
int main()
{
  char ch = next_char();

  while ( ch != '\n' ) {
        // tee jotain ...
        ch = next_char();
  }

  // ...
}
```

voidaan kirjoittaa seuraavasti:

```
extern char next_char();
int main()
{
 char ch;

 while (( ch = next_char() ) != '\n' ) {
        // tee jotain ...
 }

  // ...
}
```

Lisäsulut ovat välttämättömiä, koska sijoitusoperaattorin sidontajärjestys on alhaisempi kuin yhtäsuuruusoperaattorin. Sidontajärjestys määrää lausekkeen arviointijärjestyksen, jossa korkeammalla olevat operaattorit arvioidaan ensiksi. Ilman sulkuja testataan yhtäsuuruus

```
next_char() != '\n'
```

ensiksi ja sitten ch-olioon sijoitetaan true- tai false-arvo riippuen siitä, onko next_char():n tulos yhtäsuuri kuin rivinvaihtomerkki — aivan selvää on, että tätä emme tarkoittaneet! (Katsomme sidontajärjestystä tarkemmin kohdassa 4.13.)

Samalla tavalla voidaan sijoitusoperaattoreita yhdistellä edellyttäen, että jokaisen sijoitettavan operandin tyyppi on samaa yleistä tyyppiä. Esimerkiksi seuraavassa

```
int main()
{
   int ival, jval;
   ival = jval = 0; // ok: jokaiseen sijoitetaan 0
   // ...
}
```

ival ja jval saavat molemmat arvokseen 0. Kuitenkaan seuraava ei ole sallittu, koska ival ja pval ovat erityyppisiä olioita, vaikkakin 0 on kelvollinen arvo, joka voidaan sijoittaa kumpaankin:

```
int main()
{
   int ival; int *pval;
   ival = pval = 0; // virhe: eivät samaa tyyppiä
   // ...
}
```

Seuraava saattaa olla sallittu tai sitten ei; se ei kuitenkaan toimi määrittelynä molemmille:

```
int main()
{
   // ...
   int ival = jval = 0; // saattaa olla sallittu tai sitten ei
   // ...
}
```

Esimerkin koodi on kelvollinen ainoastaan, jos jval on määritelty aikaisemmin ja on sellainen tyyppi, johon voidaan sijoittaa nolla. Siinä tapauksessa ival alustetaan arvolla, joka on tuloksena, kun jval-olioon sijoitetaan arvo 0. Jotta määriteltäisiin molemmat oliot, pitää kirjoittaa uudelleen seuraavasti:

```
int main()
{
   // ok: määrittelee oliota ival ja jval...
   int ival = 0, jval = 0;
   // ...
}
```

Käytämme usein operaattoria olioon ja sitten sijoitamme tuloksen uudelleen olioon kuten tässä:

```
int arraySum( int ia[], int sz )
{
   int sum = 0;
   for ( int i = 0; i < sz; ++i )
        sum = sum + ia[ i ];
   return sum;
}
```

Tästä syystä on olemassa yhdistettyjä sijoitusoperaattoreita. Esimerkiksi edellinen funktio voidaan kirjoittaa uudelleen käyttäen yhdistettyä plus-sijoitusoperaattoria:

```
int arraySum( int ia[], int sz )
{
   int sum = 0;
   for ( int i = 0; i < sz; ++i )
       // on yhtäkuin: sum = sum + ia[ i ];
       sum += ia[ i ];
   return sum;
}
```

Yhdistetyn sijoitusoperaattorin yleinen syntaktinen muoto on

```
a op= b;
```

jossa op= voi olla joku seuraavista operaattoreista:

```
+=      -=      *=   /=      %=
<<=     >>=     &=   ^=      |=
```

Jokainen yhdistetty operaattori on yhtä seuraavan “pitkän kaavan” sijoituksen kanssa:

```
a = a op b;
```

Esimerkiksi vektorin summauksen pitkän kaavan mukainen merkintätapa on

```
sum = sum + ia[ i ];
```

Harjoitus 4.6

Seuraava ei ole sallittua. Miksi? Kuinka korjaisit sen?

```
int main() {
   float fval;
   int   ival;
   int  *pi;

   fval = ival = pi = 0;
}
```

Harjoitus 4.7

Vaikka seuraavat ovat sallittuja, ne eivät toimi kuten ohjelmoija olettaa. Miksi? Kuinka toteutat ne uudelleen niin kuin ohjelmoija todennäköisesti on tarkoittanut?

```
(a) if ( ptr = retrieve_pointer() != 0 )
(b) if ( ival = 1024 )
(c) ival += ival + 1;
```

4.5 Lisäys- ja vähennysoperaattorit

Lisäys- (++) ja vähennys- (--) -operaattorit tarjoavat kätevän lyhennetyn merkintätavan arvon 1 lisäämiselle tai vähentämiselle oliosta. Niitä käytetään yleisimmin indeksin, iteraattorin tai osoittimen arvon korottamiseen tai alentamiseen. Esimerkiksi:

```
#include <vector>
```

```
#include <cassert>

int main()
{
        int ia[10] = {0,1,2,3,4,5,6,7,8,9};
        vector< int > ivec( 10 );

        int ix_vec = 0, ix_ia = 9;
        while ( ix_vec < 10 )
                ivec[ ix_vec++ ] = ia[ ix_ia-- ];

        int *pia = &ia[9];
        vector<int>::iterator iter = ivec.begin();

        while ( iter != ivec.end() )
                assert( *iter++ == *pia-- );
}
```

Lauseke

```
ix_vec++
```

on lisäysoperaattorin *jälkiliitemuoto*. Se kasvattaa ix_vec-olion arvoa sen *jälkeen,* kun sen nykyistä arvoa on käytetty ivec-vektoriin. Esimerkiksi while-silmukan ensimmäisellä toistokerralla ix_vec sisältää arvon 0. Tuo arvo toimii indeksinä ivec-vektoriin. Arvoa ix_vec kasvatetaan arvolla 1 arvoon 1, mutta tätä uutta arvoa ei todellisuudessa käytetä ennen kuin seuraavalla silmukan toistokerralla. Vähennysoperaattorin jälkiliitemuoto toimii samalla tavalla. ix_ia:n nykyistä arvoa käytetään indeksinä ia-vektoriin; arvoa ix_ia vähennetään sitten arvolla 1.

Näiden kahden operaattorin *etuliitemuotoa* tuetaan myös. Etuliitemuodossa nykyistä arvoa aluksi joko kasvatetaan tai vähennetään ja arvoa käytetään vasta sitten. Täten, jos kirjoitamme

```
// väärin: menee yhden yli molempien loppupäässä
int ix_vec = 0, ix_ia = 9;
while ( ix_vec < 10 )
    ivec[ ++ix_vec ] = ia[ --ix_ia ];
```

arvoa ix_vec kasvatetaan arvolla 1 arvoon 1 ennen kuin sitä käytetään indeksinä ivec-vektoriin. Samalla tavalla arvoa ix_ia vähennetään arvolla 1 arvoon 8 ennen sen käyttöä indeksinä vektoriin ia. Jotta silmukkamme toimisi oikein, pitää asettaa kahden indeksin alkuarvoihin 1 vähemmän ja 1 enemmän kuin arvot, jotka todellisuudessa haluamme käsitellä.

```
// ok: molempien loppupäät korjattu
int ix_vec = -1, ix_ia = 10;
while ( ix_vec < 10 )
    ivec[ ++ix_vec ] = ia[ --ix_ia ];
```

Katsotaanpa viimeisenä esimerkkinä pinon suunnittelua. Pino on perustavaa laatua oleva tietokonetieteen tietoabstraktio, joka mahdollistaa arvojen sisäkkäin asettamisen ja hakemisen

LIFO-järjestyksessä — viimeisenä sisään, ensimmäisenä ulos. Pinon kaksi perustoimintoa ovat uuden arvon *työntäminen* (*push*) pinoon ja viimeisen arvon *vetäminen* (*pop*) pinosta. Oletetaan argumenttien vuoksi, että pino on toteutettu vektorina.

Pino pitää yllä top-oliota, joka ilmaisee seuraavan vapaan paikan, johon arvo voidaan työntää. Toteuttaaksemme työntämisen periaatteen, pitää sijoittaa arvo paikkaan, jonka top osoittaa, ja sitten kasvattaa top-arvoa yhdellä. Millaisen ilmentymän lisäysoperaattorista tämä vaatii? Haluamme käyttää nykyistä arvoa ja sitten kasvattaa sitä arvolla 1. Näin käyttäytyy jälkiliitemuoto:

```
stack[ top++ ] = value;
```

Toteuttaaksemme vetämisen periaatteen, pitää aluksi vähentää top-arvoa yhdellä ja sitten palauttaa paikan arvo, jota juuri vähennetty top-arvo indeksoi. Näin käyttäytyy etuliitemuoto:

```
int value = stack[ --top ];
```

(Pinoluokkamme toteutus on tämän luvun lopussa. Vakiokirjaston pinoluokkaa käsitellään kohdassa 6.16.)

Harjoitus 4.8

Mitä luulet, miksi C++:aa ei nimetty ++C:ksi?

4.6 Kompleksilukujen operaatiot

Vakiokirjaston kompleksilukuluokka on erinomainen esimerkki oliopohjaisesta luokka-abstraktiosta. Sen operaattorien ylikuormituksen käytön avulla kompleksiluokan oliotyyppejä voidaan käyttää melkein yhtä helposti kuin yksinkertaisia sisäisiä aritmeettisia tyyppejä. Kuten kohta tulemme näkemään, C++ ei tue vain tavallisia aritmeettisia operaattoreita — kuten yhteen-, vähennys-, kerto- ja jakolasku — vaan myös kompleksilukuja voidaan sekoittaa keskenään sisäisten aritmeettisten tyyppien kanssa. Ohjelmoijalle kompleksiluvut ovat osa peruskieltä, vaikka niiden toteutus onkin vakiokirjastossa. (Huomaa, että tässä luvun kohdassa käsitellään vain kompleksilukuluokan käyttöä; jos haluat tutustua kompleksilukumatematiikkaan, katso julkaisua [PERSON68] tai jotain matematiikan koulukirjaa.) Voidaan kirjoittaa esimerkiksi:

```
#include <complex>

complex< double > a;
complex< double > b;

// ... sijoita a:han ja b:hen ...

complex< double > c = a * b + a / b;
```

Kompleksityyppejä ja aritmeettisia tyyppejä voidaan sekoittaa keskenään kuten tässä:

```
complex< double > complex_obj = a + 3.14159;
```

Samalla tavalla kompleksiluku voidaan alustaa aritmeettisella tyypillä tai sen arvolla kuten tässä

```
double dval = 3.14159;
complex_obj = dval;
```

tai

```
int ival = 3;
complex_obj = ival;
```

Päinvastaista ei kuitenkaan tueta — tämä tarkoittaa, että aritmeettista tyyppiä ei voida alustaa tai siihen sijoittaa kompleksilukuluokan oliota. Esimerkiksi seuraava johtaa käännöksenaikaiseen virheeseen:

```
// virhe: implisiittistä konversiota ei ole kompleksiluvulle
// aritmeettisen tyypin rakentamiseksi

double dval = complex_obj;
```

Jotta se voitaisiin tehdä, pitää eksplisiittisesti ilmaista, minkä kompleksiolion komponentin haluamme sijoittaa. Kompleksilukuluokka tukee paria operaatiota joko reaali- tai imaginääriosan lukemista varten. Voimme esimerkiksi käyttää joko *jäsenen käsittelysyntaksia*

```
double re = complex_obj.real();
double im = complex_obj.imag();
```

tai vastaavaa jäsenetöntä syntaksia:

```
// vastaa edellistä jäsensyntaksia
double re = real( complex_obj );
double im = imag( complex_obj );
```

Kompleksilukuluokka tukee neljää yhdistettyä sijoitusoperaattoria: ne ovat lisäys (+=), vähennys (-=), kerto (*=) ja jako (/=). Täten voimme kirjoittaa esimerkiksi:

```
complex_obj += second_complex_obj;
```

Kompleksiluvulle tuetaan sekä syöttöä että tulostusta. Kompleksiluvun tulostus on pilkulla erotettu järjestetty pari sulkujen sisällä. Ensimmäinen arvo on reaaliosa ja toinen arvo on imaginääriosa. Esimerkiksi

```
complex< double > complex0( 3.14159, -2.171 );
complex< double > complex1( complex0.real() );

cout << complex0 << " " << complex1 << endl;
```

johtaa seuraavaan:

```
( 3.14159, -2.171 ) ( 3.14159, 0.0 )
```

Jokainen seuraavista numeerisista esitystavoista voidaan lukea kompleksiluvuksi:

```
// kelvollisia kompleksiluvun syöttömuotoja
// 3.14159       ==> complex( 3.14159 );
```

```
// ( 3.14159 )    ==> complex( 3.14159 );
// ( 3.14, -1.0 ) ==> complex( 3.14, -1.0 );

// voidaan lukea kuten tässä
// cin >> a >> b >> c
// jossa a, b, c ovat kompleksilukuja

3.14159  ( 3.14159 )  ( 3.14, -1.0 )
```

Kompleksilukuluokan tukemiin lisäoperaattoreihin kuuluvat sqrt(), abs(), polar(), sin(), cos(), tan(), exp(), log(), log10() ja pow().

Harjoitus 4.9

Vakiokirjaston Rogue Wave -toteutus, joka on käytettävissämme tätä kirjaa kirjoittaessamme, tukee noita neljää yhdistettyä sijoitusoperaattoria vain oikeanpuoleisen kompleksityypin operandin osalta. Esimerkiksi yritys kirjoittaa

```
complex_obj += 1;
```

johtaa käännöksenaikaiseen virheeseen, vaikka C++-standardi sanoo, että sijoitus on sallittu. (Ei ole epätavallista löytää vielä tässä vaiheessa toteutuksia, jotka ovat jääneet jälkeen C++-standardista.) Voimme korjata tämän erehdyksen tekemällä omat ilmentymät yhdistetyistä operaattoreista. Tässä on esimerkiksi malliton ilmentymä yhdistetystä sijoitusoperaattorista lisäystä varten kompleksiluvulle complex<double>:

```
#include <complex>
inline complex<double>&
operator+=( complex<double> &cval, double dval )
{
    return cval += complex<double>( dval );
}
```

Kun otamme tämän ilmentymän mukaan ohjelmaamme, tapahtuu edellinen yhdistetty lisäys-sijoitusoperaatio oikein. (Tämä on esimerkki ylikuormitetun operaattorin tekemisestä luokkatyypille. Operaattoreiden ylikuormitus käsitellään tarkemmin luvussa 15.)

Toteuta edellistä määrittelyä mallina käyttäen kolme muuta yhdistettyä operaattoria kompleksiluvulle complex<double>. Lisää ne seuraavaan pieneen ohjelmaan ja suorita se.

```
#include <iostream>
#include <complex>

// yhdistettyjen operaattoreiden määrittelyt tulevat tähän

int main()
{
    complex< double > cval( 4.0, 1.0 );

     cout << cval << 'endl';
     cval += 1;
```

```
    cout << cval << endl;
    cval -= 1;
    cout << cval << endl;
    cval *= 2;
    cout << cval << endl;
    cout /= 2;
    cout << cval << endl;
}
```

Harjoitus 4.10

C++-standardin kompleksilukutyypillä ei ole lisäysoperaattoria, vaikka sen puuttuminen ei ole luontaista kompleksiluvuille — loppujen lopuksi

```
cval += 1;
```

itse asiassa kasvattaa cval:in reaaliosaa arvolla 1. Määrittele lisäysoperaattori, lisää se seuraavaan ohjelmaan, käännä ja suorita se:

```
#include <iostream>
#include <complex>

// lisäysoperaattorin määrittelyt tulevat tähän

int main()
{
    complex< double > cval( 4.0, 1.0 );

    cout << cval << 'endl';
    ++cval;
    cout << cval << endl;
 }
```

4.7 Ehdollinen operaattori

Ehdollinen operaattori on kätevä vaihtoehto if-else-lauseille. Esimerkiksi sen sijaan, että kirjoittaisimme

```
bool is_equal = false;
if ( !strcmp( str1, str2 ))
   is_equal = true;
```

voimme kirjoittaa

```
bool is_equal = !strcmp( str1, str2 ) ? true : false;
```

Ehdollisen operaattorin syntaktinen muoto on seuraava:

```
lauseke1 ? lauseke2 : lauseke3;
```

lauseke1 arvioidaan aina, ja se johtaa joko arvoon true tai false. Jos sen arvo on true, lauseke2 arvioidaan; muussa tapauksessa arvioidaan lauseke3. Esimerkiksi seuraava

```
int min( int ia, int ib )
    { return ( ia < ib ) ? ia : ib; }
```

on yhtäpitävä lyhenne seuraavasta

```
int min( int ia, int ib ) {
   if ( ia < ib )
       return ia;
   return ib;
}
```

Seuraava ohjelma kuvaa, kuinka ehdollista operaattoria voitaisiin käyttää.

```
#include <iostream>

int main()
{
   int i = 10, j = 20, k = 30;

   cout << "Suurempi luvuista "
       << i << " ja " << j << " on "
       << ( i > j ? i : j ) << endl;

   cout << "Luku " << i << " on"
       << ( i % 2  ? " pariton." : " parillinen." )
       << endl;

   /* Ehdollisia operaattoreita voidaan laittaa sisäkkäin,
    * mutta syvää sisäkkäisyyttä on vaikea lukea.
    * Tässä esimerkissä
    * max saa arvokseen suurimman kolmesta muuttujasta.
    */

   int max = ( (i > j)
      ? (( i > k) ? i : k)
      : ( j > k ) ? j : k);

   cout << "Suurin luvuista "
       << i << ", " << j << " ja " << k
       << " on " << max << endl;
}
```

Kun ohjelma käännetään ja suoritetaan, ohjelma generoi seuraavan tulostuksen:

```
Suurempi luvuista 10 ja 20 on 20
Luku 10 on parillinen
Suurin luvuista 10, 20 ja 30 on 30
```

4.8 Sizeof-operaattori

Operaattori sizeof palauttaa olion tai tyyppinimen koon tavuina. Sitä käytetään seuraavissa muodoissa:

```
sizeof (tyyppinimi );
sizeof ( olio );
sizeof olio;
```

Palautettu arvo on tyyppiä size_t, joka on konekohtainen typedef-nimi ja löytyy cstddef-otsikkotiedostosta. Tässä on esimerkki, kuinka sizeof-operaattorin molempia muotoja käytetään:

```
#include <cstddef>

int ia[] = { 0, 1, 2 };

// sizeof palauttaa koko taulukon koon
size_t array_size = sizeof ia;

// sizeof palauttaa int-tyypin koon
size_t element_size = array_size / sizeof( int );
```

Kun sizeof-operaattoria käytetään taulukkoon, kuten ia-taulukkoon edellisessä esimerkissä, se palauttaa koko taulukon koon tavuina eikä yksinkertaisesti sen ensimmäisen elementin kokoa tai sisältämiensä elementtien lukumäärää. Esimerkiksi koneissa, joissa int on 4 tavua pitkä, sizeof ilmaisee, että ia-taulukon koko on 12 tavua. Samalla tavalla, kun kirjoitamme

```
int *pi = new int[ 3 ];
size_t pointer_size = sizeof ( pi );
```

sizeof(pi):n palauttama arvo on osoittimen koko tavuina int-tyyppiin, eikä taulukko, jota pi osoittaa.

Tässä on pieni funktio, jossa kokeillaan sizeof()-operaattorin käyttöä:

```
#include <string>
#include <iostream>
#include <cstddef>

int main()
{
   size_t ia;

   ia = sizeof( ia ); // ok
   ia = sizeof ia;    // ok

   // ia = sizeof int; // virhe
   ia = sizeof( int ); // ok

   int *pi = new int[ 12 ];

   cout << "pi: " << sizeof( pi )
        << " *pi: " << sizeof( *pi )
        << endl;

   // string:in koko on riippumaton
   // merkkijonon pituudesta, johon se osoittaa
   string st1( "foobar" );
   string st2( "a mighty oak" );

   string *ps = &st1;

   cout << "st1: "  << sizeof( st1 )
        << " st2: " << sizeof( st2 )
        << " ps: "  << sizeof( ps )
        << " *ps: " << sizeof( *ps )
        << endl;

   cout << "short :\t"    << sizeof(short)    << endl;
   cout << "short* :\t"   << sizeof(short*)   << endl;
   cout << "short& :\t"   << sizeof(short&)   << endl;
   cout << "short[3] :\t" << sizeof(short[3]) << endl;
}
```

Kun ohjelma käännetään ja suoritetaan, ohjelma generoi seuraavan tulostuksen:

```
pi: 4 *pi: 4
st1: 12 st2: 12 ps: 4 *ps: 12
short : 2
short* : 4
short& : 2
short[3] : 6
```

Kuten ohjelmaesimerkki osoittaa, jos sizeof-operaattoria käytetään osoitintyyppiin, se palauttaa muistin koon, joka tarvitaan tuolle tyypille. Sen sijaan, jos sizeof-operaattoria käytetään viittaustyyppiin, se palauttaa muistin koon, joka tarvitaan viitatulle oliolle.

Kun sizeof-operaattoria käytetään char-tyypille, taataan kaikissa C++-toteutuksissa, että tulos on 1.

```
// on taattu kaikissa toteutuksissa, että tämän arvo on 1
size_t char_size = sizeof( char );
```

Operaattori sizeof arvioidaan käännöksen aikana ja siten sitä pidetään vakiolausekkeena. Sen mukaisesti sitä voidaan käyttää aina siellä, missä vakiolauseketta vaaditaan, kuten taulukon ulottuvuudessa tai mallin tyypittömänä parametrina. Esimerkiksi:

```
// ok: käännöksenaikainen vakiolauseke
int array[ sizeof( some_type_T )];
```

4.9 New- ja delete-lausekkeet

Jokaiselle ohjelmalle on varattu varasto vapaasti käytettävää muistia, jota se voi käyttää ohjelman suorituksen aikana. Tätä käytettävissä olevaa muistivarastoa kutsutaan ohjelman *vapaavarastoksi* (*free store*) eli *keoksi* (*heap*). Suorituksenaikaista muistinvarausta kutsutaan *dynaamiseksi muistinvaraukseksi*. Kuten näimme luvussa 1, tämä toteutetaan käyttämällä new-lauseketta joko sisäiselle tyypille tai käyttäjän määrittelemälle luokkatyypille. Lauseke new palauttaa osoittimen juuri varattuun olioon. Esimerkiksi

```
int *pi = new int;
```

varaa yhden int-tyyppisen olion vapaavarastosta ja alustaa pi:n sen osoitteella. Varsinainen varattu olio vapaavarastossa on alustamaton. Voimme määrittää alkuarvon kuten seuraavassa:

```
int *pi = new int( 1024 );
```

Tämä ei ainoastaan varaa oliota, vaan myös alustaa sen arvolla 1024.

Jos haluamme varata dynaamisesti taulukon olioita, kirjoitamme

```
int *pia = new int[ 10 ];
```

Tämä varaa taulukon kymmenelle int-tyyppiselle oliolle vapaavarastosta ja alustaa pia:n sen osoitteella. Taulukon elementit ovat kuitenkin alustamattomia. Ei ole olemassa syntaksia, joka määrittäisi eksplisiittisen joukon alkuarvoja dynaamisesti varatun taulukon elementeille. (Kun on kyse luokkaolioiden taulukosta, oletusmuodostajaa (jos määritelty) käytetään jokaisen elementin kohdalla vuorollaan.) Esimerkiksi

```
string *ps = new string;
```

varaa yhden string-luokkaolion vapaavarastosta ja alustaa ps:n sen osoitteella ja käynnistää sitten string-luokkaolion oletusmuodostajan. Samalla tavalla

```
string *psa = new string[ 10 ];
```

varaa kymmenen string-luokkaolion taulukon ja alustaa psa:n sen osoitteella ja käynnistää sitten string-luokan oletusmuodostajan jokaiselle taulukon elementille erikseen.

Eräs puoli vapaavarastomuistissa on: sieltä varatut oliot ovat nimettömiä. Lauseke new ei palauta varsinaista varattua oliota, vaan varatun olion osoitteen. Kaikki olion käsittely tapahtuu epäsuorasti tuon osoitteen kautta.

Kun oliomme käyttö on päättynyt, pitää eksplisiittisesti palauttaa olion muistitila vapaavarastoon. Teemme sen käyttämällä delete-lauseketta osoittimelle olioon, joka alun perin varattiin new-lausekkeen kautta. (Lauseketta delete ei tulisi käyttää osoittimeen, joka osoittaa muistialueelle, jota ei ole varattu new-lausekkeella.) Esimerkiksi

```
delete pi;
```

vapauttaa pi:n osoittaman int-olion muistitilan vapaavarastoon. Samalla tavalla

```
delete ps;
```

vapauttaa string-luokkaolion muistitilan, johon ps osoittaa, käyttäen siihen ensiksi string-luokan tuhoajaa ja sitten palauttamalla sen takaisin vapaavarastoon. Lopuksi

```
delete [] pia;
```

vapauttaa pia:n osoittaman kymmenen int-olion taulukon muistitilan vapaavarastoon. Tyhjä hakasulkupari delete-avainsanan ja taulukon osoittimen välillä edustaa erityistä delete-lausekkeen syntaksia taulukon sen muistitilan vapauttamiseksi, jonka varaaminen on tapahtunut new-lausekkeen kautta.

Luvussa 8 katsotaan dynaamista muistinvarausta ja new- sekä delete-lausekkeiden käyttöä yksityiskohtaisesti.

Harjoitus 4.11

Mitkä seuraavista eivät ole sallittuja tai ovat virheellisiä, vai onko yksikään?

```
(a) vector<string> svec( 10 );
(b) vector<string> *pvec1 = new vector<string>(10);
(c) vector<string> **pvec2 = new vector<string>[10];
(d) vector<string> *pv1 = &svec;
(e) vector<string> *pv2 = pvec1;

(f) delete svec;
(g) delete pvec1;
(h) delete [] pvec2;
(i) delete pv1;
(j) delete pv2;
```

4.10 Pilkkuoperaattori

Pilkkulauseke on joukko lausekkeita, jotka on eroteltu toisistaan pilkuin. Nämä lausekkeet arvioidaan vasemmalta oikealle. Pilkkulausekkeen tulos on oikeanpuoleisimman lausekkeen arvo. Seuraavassa esimerkissä ehdollisen operaattorin molemmat puolet ovat pilkkulausekkeita. Ensimmäisen pilkkulausekkeen arvo on ix; toisen arvo on 0.

```
int main()
{
   // esimerkkejä pilkkulausekkeesta
   // ia, sz ja index on määritelty jossain muualla ...
   int ival = (ia != 0)
            ? ix=get_value(), ia[index]=ix
            : ia=new int[sz], ia[index ]=0;
   // ...
}
```

4.11 Biteittäiset operaattorit

Taulukko 4.3 Biteittäiset operaattorit.

Operaattori	Toiminto	Käyttö
~	biteittäinen EI	~lauseke
<<	siirto vasemmalle	lauseke1 << lauseke2
>>	siirto oikealle	lauseke1 >> lauseke2
&	biteittäinen JA	lauseke1 & lauseke2
^	biteittäinen POIS SULKEVA TAI	lauseke1 ^ lauseke2
|	biteittäinen TAI	lauseke1 | lauseke2
&=	biteittäinen JA sijoitus	lauseke1 &= lauseke2
^=	biteittäinen POIS SULKEVA TAI sijoitus	lauseke1 ^= lauseke2
|=	biteittäinen TAI sijoitus	lauseke1 |= lauseke2

Biteittäinen (*bitwise*) operaattori tulkitsee operandinsa järjestettynä bittikokoelmana joko yksittäin tai kentiksi ryhmitettyinä. Jokainen bitti voi sisältää joko arvon 0 (pois päältä) tai 1 (päällä). Biteittäinen operaattori sallii ohjelmoijan testata tai asettaa yksittäisiä bittejä tai bittikenttiä. Oliota, jota käytetään erillisenä yksittäisten bittien kokoelmana, kutsutaan *bittivektoriksi*. Bittivektorit ovat tiivis menetelmä pitää yllä kyllä/ei-tietoa (joskus nimitetään *lipuiksi* (*flags*)) kohdejoukolle tai tilanteille. Esimerkiksi kääntäjissä tallennetaan joskus tyyppimäärit-

telyjen määreet kuten const ja volatile bittivektoriin. Iostream-kirjasto käyttää bittivektoreita muotoilutilan esittämiseen, kuten esimerkiksi, tulisiko kokonaistyypit tulostaa desimaalisena, heksadesimaalisena vai oktaalisena.

Aivan kuten C++-standardissa on olemassa kaksi tapaa tukea merkkijonoja (string-luokkatyyppi ja C-tyylinen merkkijono) ja elementtien järjestettyä kokoelmaa (vektorin malliluokka ja sisäinen taulukkotyyppi) on olemassa kaksi lähestymistapaa bittivektorien tukemiseen. C++:n esistandardissa ja C-kielessä bittivektori esitetään käyttäen sisäistä kokonaistyyppiä, joka tyypillisesti on unsigned int. Olio toimii bittisäiliönä ja ohjelmoija käsittelee sen sisältöä tässä esitetyillä biteittäisillä operaatioilla. Vakiokirjastossa on bittijoukkoluokka, joka tukee bittivektorin luokka-abstraktiota. Bittijoukko-olio kapseloi sisäänsä bittivektorin yksityiskohdat ja se vastaa kyselyihin kuten “Onko yhtään biteistäsi asetettuna?”, “Kuinka monta biteistäsi on asetettu?” ja siinä on myös joukko operaatioita bittien asettamiseen, uudelleenasettamiseen ja testaamiseen.

Yleensä suosittelemme vakiokirjaston luokka-abstraktioiden käyttöä — tässä tapauksessa bittijoukkoluokan käyttöä kokonaistyyppien suoran biteittäisen käsittelyn asemesta. Mielestämme kuitenkin molempien esitystapojen tuntemus on välttämätöntä, koska todennäköistä on, että joudut lukemaan tai ehkä muokkaamaan olemassaolevaa koodia. Täydellisyyden vuoksi esitämme molemmat tavat. Tämän osan loppuun saakka katsomme sisäisten kokonaistyyppien käyttöä bittivektoreina ja biteittäisten operaattoreiden käyttöä. Seuraavassa kohdassa katsomme bittijoukkoluokkaa.

Kun käytämme kokonaistyyppiä bittivektorina, tyyppi voi olla joko etumerkillinen tai etumerkitön; suosittelemme ehdottomasti etumerkitöntä tyyppiä. Se, kuinka “etumerkkibitti” käsitellään lukuisissa biteittäisissä operaatioissa, on tuntematon, ja siten todennäköisesti poikkeaa eri toteutuksissa. Ohjelmat, jotka toimivat jossain toteutuksessa, voivat toimia virheellisesti toisessa.

Biteittäinen EI-operaattori (~) kääntää operandinsa bitit. Jokainen 1-bitti asetetaan 0-bitiksi ja jokainen 0-bitti asetetaan 1-bitiksi.

Biteittäiset siirto-operaattorit (<<, >>) siirtävät vasemmanpuoleisen operandin bittejä tietyn positiomäärän joko vasemmalle tai oikealle. Operandin liiat bitit hävitetään. Siirto vasemmalle -operaattori (<<) lisää 0-arvoisia bittejä oikealta. Siirto oikealle -operaattori (>>) lisää 0-arvoisia bittejä vasemmalta, jos operandi on etumerkitön. Jos operandi on etumerkillinen, se voi lisätä joko etumerkin kopioita tai 0-arvoisia bittejä. Jälleen, tämä käyttäytyminen on määritelty toteutukseen.

Biteittäinen JA-operaattori (&) tarvitsee kaksi kokonaisoperandia. Jokaisessa bittipositiossa tulos on 1, jos molemmat operandit sisältävät 1-bitin; muussa tapauksessa tulos on 0. (Tätä operaattoria ei tulisi sekoittaa loogiseen JA-operaattoriin (&&); valitettavasti näyttää siltä, että jokainen joskus tekee juuri niin.)

Biteittäinen POIS SULKEVA -operaattori (^) tarvitsee kaksi kokonaisoperandia. Jokaisessa bittipositiossa tulos on 1, jos jompikumpi operandi sisältää 1-bitin, mutta ei molemmat; muussa tapauksessa tulos on 0.

Biteittäinen TAI (mukaan ottava tai) -operaattori (|) tarvitsee kaksi kokonaisoperandia. Jokaisessa bittipositiossa tulos on 1, jos jompikumpi tai molemmat operandit sisältävät 1-bitin; muussa tapauksessa tulos on 0. (Tätä operaattoria ei tulisi sekoittaa loogiseen TAI-operaattoriin (||).)

Katsokaamme yksinkertaista esimerkkiä. Opettajalla on 30 oppilasta luokassa. Joka viikko luokalle pidetään koe, jonka tulos voidaan hyväksyä tai hylätä. Jokaisen kokeen tuloksia pidetään yllä bittivektorissa. (Huomaa, että bitit numeroidaan alkaen arvosta 0. Täten 1-bitti todellisuudessa edustaa toista bittiä. Esimerkissämme tuhlaamme ensimmäisen bitin puhuaksemme bitistä 1 ensimmäisenä bittinä jne. Manittakoon, että opettajamme ei ole opiskellut tietotekniikkaa.)

```
unsigned int quiz1 = 0;
```

Opettajan pitää pystyä laittamaan yksittäisiä bittejä päälle ja pois päältä. Esimerkiksi opiskelija numero 27 on tehnyt kokeen ja se on hyväksytty. Opettajan pitää laittaa bitti numero 27 päälle. Ensimmäinen tehtävä on asettaa kokonaisluvun 27. bitti arvoon 1, kun taas muut bitit jäävät arvoon 0. Tämä voidaan tehdä siirto vasemmalle -operaattorilla ja kokonaislukuvakiolla 1:

```
1 << 27;
```

Jos tätä arvoa käytetään quiz1:een biteittäisellä TAI-operaattorilla, kaikki muut paitsi 27. bitti jäävät ennalleen eli 27. bitti tulee päälle:

```
quiz1 |= 1<<27;
```

Kuvitellaan, että opettaja tutkii kokeen uudelleen ja huomaa, että opiskelija numero 27 oli itse asiassa epäonnistunut kokeessa. Opettajan pitää nyt laittaa 27. bitti pois päältä. Tällä kertaa kokonaisluvun kaikki muut paitsi 27. bitti pitää laittaa päälle. Huomaa, että tämä on kääntäen verrannollinen edelliseen kokonaislukuun. Käyttämällä biteittäistä EI-operaattoria edelliseen kokonaislukuun voidaan laittaa kaikki muut paitsi 27. bitti päälle:

```
~( 1<<27 );
```

Jos tätä arvoa käytetään quiz1:een biteittäisellä JA-operaattorilla, kaikki muut paitsi 27. bitti jäävät ennalleen eli 27. bitti tulee pois päältä:

```
quiz1 &= ~(1<<27);
```

Tästä nähdään, kuinka opettaja voi päätellä bitin päällä tai pois päältä olon tilan. Mietitäänpä jälleen opiskelijaa numero 27 (todellisuudessa hänen nimensä on Anna). Ensimmäinen tehtävä on asettaa kokonaisluvun 27. bitti arvoon 1. Tämän arvon biteittäinen JA yhdessä quiz1:n kanssa arvioidaan arvoksi true, jos quiz1:n 27. bitti on myös päällä; muussa tapauksessa sen arvoksi tulee false:

```
bool hasPassed = quiz1 & (1<<27);
```

Koska biteittäiset operaattorit ovat virhealttiita matalan tason bittikäsittelystä johtuen, ne kapseloidaan tyypillisesti joko esikääntäjän makroihin tai välittömiin funktioihin. Esimerkiksi:

```
inline bool bit_on( unsigned int ui, int pos )
{
   return ui & (1 << pos );
}
```

Tämä saatetaan sitten käynnistää seuraavasti:

```
enum students { Danny = 1, Jeffrey, Ethan, Zev, Ebie, // ...
                AnnaP = 26, AnnaL = 27 };
const int student_size = 27;

// aloittaa tarkoituksella kohdasta 1
bool has_passed_quiz[ student_size+1 ];
for ( int index = 1; index <= student_size; ++index )
    has_passed_quiz[ index ] = bit_on( quiz1, index );
```

Tietysti, kun aloitamme suoran käytön kapseloinnin, on seuraava looginen askel kapseloida koko bittivektorin merkintätapa — vakiokirjaston tapauksessa se on bittijoukon luokka-abstraktio. Sattumalta tämä on seuraavien kohtien pääaiheena.

Harjoitus 4.12

Oletetaan seuraavat kaksi määrittelyä:

```
unsigned int ui1 = 3, ui2 = 7;
```

Mikä on tuloksena jokaisesta seuraavasta lausekkeesta?

(a) ui1 & ui2 (c) ui1 | ui2
(b) ui1 && ui2 (d) ui1 || ui2

Harjoitus 4.13

Käytä mallina välitöntä funktiota bit_on() ja toteuta bit_turn_on() (laittaa tietyn bitin päälle), bit_turn_off() (ottaa tietyn bitin pois päältä), flip_bit() (vaihtaa tietyn bitin arvon päinvastaiseksi) ja bit_off() (testaa, onko tietty bitti pois päältä). Tämä joukko on välittömiä funktioita, jotka käsittelevät unsigned int -tyypistä bittivektoria. Kirjoita sitten pieni ohjelma, jolla voit kokeilla funktioita.

Harjoitus 4.14

Mitä heikkouksia on harjoituksen 4.13 funktioiden eksplisiittisessä koodauksessa, jotka toimivat unsigned int -tyypillä? Eräs vaihtoehto on käyttää typedef-nimeä. Toinen vaihtoehto on käyttää mallimekanismia, joka esiteltiin kohdassa 2.5. Kirjoita välitön funktio bit_on() uudelleen käyttäen typedef-nimeä ja sitten funktion mallimekanismia.

4.12 Bittijoukko-operaatiot

Taulukko 4.4 Bittijoukko-operaatiot

Operaatio	Toiminto	Käyttö
test(pos)	onko pos. bitti päällä?	a.test(4)
any()	onko yksikään bitti päällä?	a.any()
none()	eikö yksikään bitti ole päällä?	a.none()
count()	päällä olevien bittien lukumäärä	a.count()
size()	bittielementtien lukumäärä	a.size()
[pos]	käsittele pos. bittiä	a[4]
flip()	käännä kaikki bitit	a.flip()
flip(pos)	käännä pos. bitti	a.flip(4)
set()	laita kaikki bitit päälle	a.set()
set(pos)	laita pos. bitti päälle	a.set(4)
reset()	ota kaikki bitit pois päältä	a.reset()
reset(pos)	ota pos. bitti pois päältä	a.reset(4)

Ongelma bittivektoreiden esittämisessä kokonaistyypeillä on lausekkeiden matalan tason monimutkaisuudessa, joissa käytetään biteittäisiä operaattoreita yksittäisten bittien asettamiseen, uudelleenasettamiseen ja testaamiseen. Kun esimerkiksi laitamme kokonaistyyppimme 27. bitin päälle, kirjoitamme seuraavasti:

```
quiz1 |= 1<<27;
```

Jos haluamme tehdä sen bittijoukolla, kirjoitamme joko

```
quiz1[ 27 ] = 1;
```

tai

```
quiz1.set( 27 );
```

(Kuten huomautimme aikaisemmin, bittien numerointi alkaa nollasta: käytännössä 27. bitti viittaa bittiin 28. Esimerkissämme tuhlaamme ensimmäisen bitin niin, että viittauksemme bitteihin alkaa arvosta 1.)

Kun käytämme bittijoukkoluokkaa, pitää ottaa mukaan siihen liittyvä otsikkotiedosto:

```
#include <bitset>
```

Bittijoukko voidaan esitellä kolmella eri tavalla. Oletusmäärittelyssä ilmaisemme yksin-

kertaisesti, kuinka suuren bittivektorin haluamme. Esimerkiksi

```
bitset< 32 > bitvec;
```

esittelee bittijoukko-olion, joka sisältää 32 bittiä, jotka on numeroitu väliltä 0 — 31. Oletusarvoisesti kaikki bitit alustetaan nolliksi. Kun haluamme testata, onko bittijoukko-olion biteistä yksikään asetettu, käytämme any()-operaatiota; any() palauttaa arvon true, jos yksi tai useampi bittijoukko-olion biteistä on päällä. Vektorille bitvec

```
bool is_set = bitvec.any();
```

saadaan tietysti arvo false. Vastaavasti operaatio none() palauttaa arvon true, jos bittijoukko-olion kaikki bitit ovat asetettuina nolliksi. Vektorille bitvec

```
bool is_not_set = bitvec.none();
```

saadaan arvo true. Vaihtoehtoisesti count()-operaatio palauttaa asetettujen bittien lukumäärän:

```
int bits_set = bitvec.count();
```

On olemassa kaksi tapaa asettaa yksittäinen bitti. Joko set()-operaatiolla tai indeksioperaattoria käyttäen. Esimerkiksi seuraava for-silmukka asettaa päälle jokaisen parillisen bitin:

```
for ( int index = 0; index < 32; ++index )
    if ( index % 2 == 0 )
       bitvec[ index ] = 1;
```

Samalla tavalla on olemassa kaksi tapaa testata, onko yksittäinen bitti asetettu vai ei. Operaatiolle test() annetaan bittipositio argumenttina ja se palauttaa joko arvon true tai false. Esimerkiksi:

```
if ( bitvec.test( 0 ))
   // meidän bitvec[ 0 ] toimi!;
```

Jälleen voimme käyttää vaihtoehtoisesti indeksioperaattoria:

```
cout << "bitvec: seuraavat positiot asetettu päälle:\n\t";
for ( int index = 0; index < 32; ++index )
          if ( bitvec[ index ] )
              cout << index << " ";
cout << endl;
```

Kun haluamme asettaa yksittäisen bitin pois päältä, voimme käyttää joko reset()-operaatiota tai indeksioperaattoria. Molemmat seuraavista operaatioista asettavat bitvec-bittivektorin ensimmäisen bitin pois päältä:

```
// samanarvoiset; aseta pois päältä ensimmäinen bitti
bitvec.reset( 0 );
bitvec[ 0 ] = 0;
```

Operaatioita aset() ja reset() voidaan myös käyttää vastaavasti koko bittijoukko-olion asettamiseksi päälle tai pois päältä. Se tehdään käynnistämällä haluttu operaatio välittämättä erityistä positiota. Esimerkiksi:

```
// alusta uudelleen kaikki bitit arvoon 0.
```

```
bitvec.reset();
if ( bitvec.none() != true )
      // hups! jotain on pielessä

// aseta kaikki bitit arvoon 1
bitvec.set();
if ( bitvec.any() != true )
      // hups! jälleen joitain on pielessä
```

Operaatio flip() kääntää yksittäisen bitin tai koko bittijoukko-olion päinvastaiseksi:

```
bitvec.flip( 0 ); // kääntää ensimmäisen bitin arvon päinvastaiseksi
bitvec[0].flip(); // myös kääntää ensimmäisen bitin päinvastaiseksi!

bitvec.flip();   // kääntää kaikkien bittien arvot päinvastaiseksi
```

On olemassa kaksi lisätapaa muodostaa bittijoukko-olio. Molemmat antavat mahdollisuuden alustaa yksittäiset bittipositiot arvoon 1. Toinen niistä on antaa eksplisiittisesti etumerkitön arvo argumenttina muodostajalle. Bittijoukko-olion ensimmäiset N bittipositiota alustetaan vastaavilla argumentin bittiarvoilla. Esimerkiksi

```
bitset< 32 > bitvec2( 0xffff );
```

saa aikaan sen, että bittivektorin bitvec2 alemman pään 16 bittiä asetetaan arvoon 1:

```
00000000000000001111111111111111
```

Seuraava bittivektorin bitvec3 määritys

```
bitset< 32 > bitvec3( 012 );
```

asettaa bittipositiot 1 ja 3 arvoon 1 (edellyttäen, että laskemme bittipositioiden alkavan nollasta):

```
00000000000000000000000000001010
```

Vaihtoehtoisesti voimme muodostaa bittijoukko-olion välittämällä merkkijonoargumentin, jossa on esitettynä joukko nollia ja ykkösiä kuten seuraavassa:

```
// samanarvoinen alustus kuten bitvec3:lla
string bitval( "1010" );
bitset< 32 > bitvec4( bitval );
```

Sekä bitvec4:n että bitvec3:n bittipositiot 1 ja 3 on asetettu arvoon 1, kun taas jäljelle jäävät bittipositiot ovat asetettuna arvoon 0.

Vaihtoehtoisesti voimme merkitä alueen merkkijonon elementeistä, jotka alustavat bittijoukon. Esimerkiksi seuraavassa

```
// aloita positiosta 6 pituudella 4: 1010
string bitval( "11111101011000011010101" );
bitset< 32 > bitvec5( bitval, 6, 4 );
```

bitvec5 alustetaan bittipositioistaan 1 ja 3 arvolla 1, kun taas jäljelle jäävät bittipositiot asetetaan arvoon 0 samoin kuin bittivektoreilla bitvec3 ja bitvec4. Jos jätämme pois kolmannen parametrin, joka ilmaisee aluetta merkkaavan merkkien pituuden, alue muodostuu ilmaistusta positiosta merkkijonon loppuun. Esimerkiksi:

```
// aloita positiosta 6 ja jatka merkkijonon loppuun: 1010101
string bitval( "11111101011000011010101" );
bitset< 32 > bitvec6( bitval, 6 );
```

Bittijoukkoluokka tukee kahta jäsenfunktiota, jotka voivat konvertoida bittijoukko-olion toisentyyppiseksi. Niistä toisella voimme konvertoida minkä tahansa bittijoukko-olion merkkijonon esitystapaan käyttämällä to_string()-operaatiota:

```
string bitval( bitvec3.to_string() );
```

Toisessa tapauksessa voimme konvertoida minkä tahansa bittijoukko-olion kokonaisesitystapaan unsigned long -tyypiksi käyttämällä to_long()-operaatiota, edellyttäen, että kyseinen bittijoukko voidaan esittää unsigned long -tyyppisenä. Tämä on erityisen hyödyllistä, jos pitää välittää bittijoukko C-standardin tai C++-esistandardin mukaiseen ohjelmaan.

Bittijoukkoluokka tukee biteittäisten operaattoreiden käyttöä. Esimerkiksi

```
bitset<32> bitvec7 = bitvec2 & bitvec3;
```

alustaa bitvec5:n kahden bittivektorin biteittäisellä JA:lla, kun taas

```
bitset<32> bitvec8 = bitvec2 | bitvec3;
```

alustaa bitvec6:n biteittäisellä TAI:lla. Myös biteittäisiä yhdistettyjä sijoitusoperaattoreita ja siirto-operaattoreita tuetaan (näitä käsiteltiin edellisissä kohdissa).

Harjoitus 4.15

Mitkä seuraavista bittijoukko-olioiden esittelyistä ovat virheellisiä, vai onko yksikään?

```
(a) bitset<64> bitvec(32);
(b) bitset<32> bv( 1010101 );
(c) string bstr; cin >> bstr; bitset<8>bv( bstr );
(d) bitset<32> bv; bitset<16> bv16( bv );
```

Harjoitus 4.16

Mitkä seuraavista bittijoukon käyttötavoista ovat virheellisiä, vai onko yksikään?

```
extern void bitstring(const char*);
bool bit_on(unsigned long, int );
bitset<32> bitvec;

(a) bitstring( bitvec.to_string().c_str() );
(b) if ( bit_on( bitvec.to_long(), 64 )) ...
(c) bitvec.flip( bitvec.count() );
```

Harjoitus 4.17

Mietitäänpä jonoa 1,2,3,5,8,13,21. Kuinka alustaisimme bittijoukon bitset<32>, joka edustaisi tätä jonoa? Vaihtoehtoisesti kirjoita pieni ohjelma, jolle annetaan tyhjä bittijoukko ja jonka vastaavat bitit asetetaan päälle.

4.13 Sidontajärjestys

Operaattorin sidontajärjestys (*precedence*) tarkoittaa järjestystä, jolla yhdistetyn lausekkeen operaattorit arvioidaan. Mikä on esimerkiksi lopullinen tulos, joka sijoitetaan ival:iin?

```
int ival = 6 + 3 * 4 / 2 + 2;
```

Puhtaasti vasemmalta oikealle arviointi johtaa arvoon 20. Muihin mahdollisiin tuloksiin kuuluvat 9, 14 ja 36. Mikä niistä todellisuudessa asetetaan ival:iin? 14.

C++:ssa kerto- ja jakolaskulla on korkeampi sidontajärjestys kuin yhteenlaskulla. Se merkitsee, että ne arvioidaan ensiksi. Koska kertolaskulla ja jakolaskulla on sama sidontajärjestys, arvioidaan ne kuitenkin vasemmalta oikealle. Lausekkeen arviointi on siitä syystä seuraava:

```
1. 3 * 4 => 12
2. 12 / 2 => 6
3. 6 + 6 => 12
4. 12 + 2 => 14
```

Seuraava while-silmukan ehtotesti käyttäytyy kovin eri tavalla kuin ohjelmoija oli tarkoittanut ja syynä on sijoitusoperaattorin alhaisempi sidontajärjestys kuin yhtäsuuruusoperaattorilla:

```
while ( ch = nextChar() != '\n' )
```

Ohjelmoijan aikomus on sijoittaa ch-olioon seuraava merkki ja sitten testata, onko se mahdollisesti null-merkki. Lausekkeen tarkoitus on kuitenkin testata seuraavaa merkkiä, onko se mahdollisesti null-merkki. Olioon ch sijoitetaan sitten testin totuusarvo. Seuraavaa merkkiä ei koskaan sijoiteta.

Sidontajärjestys voidaan ohittaa sulkuja käyttämällä, joka merkitsee alilausekkeita. Yhdistetyn lausekkeen arvioinnissa ensimmäinen toimenpide on arvioida kaikki välissä olevat alilausekkeet. Jokainen alilauseke korvataan tuloksellaan ja arviointi jatkuu. Sisemmät sulut arvioidaan ennen ulompia sulkuja. Esimerkiksi:

```
4 * 5 + 7 * 2 ==> 34
4 * ( 5 + 7 * 2 ) ==> 76
4 * ( (5 + 7) * 2 ) ==> 96
```

Tässä on while-silmukkamme lauseke asianmukaisesti sulkujen kanssa ilmaisemassa ohjelmoijan aikomuksia:

```
while ( (ch = nextChar()) != '\n' )
```

Operaattoreilla on sekä sidontajärjestys että assosiatiivisuus. Esimerkiksi sijoitusoperaat-

tori on oikealle assosiatiivinen. Yhteenliitetty sijoituslauseke

```
ival = jval = kval = lval // oikealle assosiatiivinen
```

sijoittaa aluksi lval:in kval:iin, sitten sen tuloksen jval:iin ja lopuksi sen tuloksen ival:iin. Aritmeettiset operaattorit ovat toisaalta vasemmalle assosiatiivisia. Lauseke

```
ival + jval + kval + lval // vasemmalle assosiatiivinen
```

lisää ival:in ja jval:in, sitten kval:in ja lopuksi lval:in.

Taulukossa 4.4 esitellään koko C++-operaattoreiden joukko sidontajärjestyksessä. Taulukon solussa olevilla operaattoreilla on keskenään sama sidontajärjestys. Yhden solun kaikilla operaattoreilla on korkeampi sidontajärjestys kuin niiden alapuolella olevan solun operaattoreilla. Esimerkiksi kerto- ja jako-operaattoreilla on sama sidontajärjestys. Niillä molemmilla on suurempi sidontajärjestys kuin yhdelläkään vertailuoperaattorilla.

Harjoitus 4.18

Yksilöi taulukkoa 4.4 käyttäen seuraavien yhdistettyjen lausekkeiden arviointijärjestys:

(a) `! ptr == ptr->next`
(b) `~ uc ^ 0377 & ui << 4`
(c) `ch = buf[ bp++ ] != '\n'`

Harjoitus 4.19

Harjoituksen 4.18 kaikki kolme lauseketta arvioidaan vastoin ohjelmoijan aikomuksia. Käytä niihin sulkuja, jotta ne arvioitaisiin siinä järjestyksessä, kuin ohjelmoija oli tarkoittanut.

Harjoitus 4.20

Seuraavien kahden lausekkeen käännös ei onnistu ja syynä on operaattorin sidontajärjestys. Selitä taulukkoa 4.4 käyttäen, miksi. Kuinka korjaisit ne?

(a)
```
int i = doSomething(), 0;
```
(b)
```
cout << ival % 2 ? "pariton" : "parillinen";
```

Taulukko 4.4 Operaattorin sidontajärjestys

Operaattori	Toiminto	Käyttö
::	globaali näkyvyys	::nimi
::	luokan näkyvyys	luokka::nimi
::	nimiavaruuden näkyvyys	nimiavaruus::nimi
.	jäsenen valitsijat	olio.jäsen
->	jäsenen valitsijat	osoitin->jäsen
[]	indeksi	muuttuja[lauseke]
()	funktion kutsu	nimi(lauseke_luettelo)
()	tyypin muodostus	tyyppi(lauseke_luettelo)
++	loppuliitelisäys	lvalue++
--	loppuliitevähennys	lvalue--
typeid	tyypin tunnus	typeid(tyyppi)
typeid	ajonaikainen tyypin tunnus	typeid(lauseke)
const_cast	tyyppikonversio	const_cast<tyyppi>(lauseke)
dynamic_cast	tyyppikonversio	dynamic_cast<tyyppi>(lauseke)
reinterpret_cast	tyyppikonversio	reinterpret_cast<tyyppi>(lauseke)
static_cast	tyyppikonversio	static_cast<tyyppi>(lauseke)
sizeof	olion koko	sizeof lauseke
sizeof	tyypin koko	sizeof(tyyppi)
++ --	etuliitelisäys	++lvalue
-- --	etuliitevähennys	--lvalue
~	biteittäinen EI	~lauseke
!	looginen EI	!lauseke
-	unaarinen miinus	-lauseke
+	unaarinen plus	+lauseke
*	käänteinen viittaus	*lauseke
&	osoite jostakin	&lauseke
()	tyyppikonversio	(tyyppi) lauseke
new	varaa olio	new tyyppi
new	varaa/alusta olio	new tyyppi(lauseke_luettelo)
new	varaa/sijoita olio	new (lauseke_luet) tyyppi(lauseke_luet)
new	varaa taulukko	kaikki muodot
delete	vapauta olio	kaikki muodot

Operaattori	Toiminto	Käyttö
delete	vapauta taulukko	kaikki muodot
->*	osoitin jäsenen valintaan	osoitin->*osoitin_jäseneen
.*	osoitin jäsenen valintaan	olio.*osoitin_jäseneen
*	kerto	lauseke * lauseke
/	jako	lauseke / lauseke
%	modulo (jakojäännös)	lauseke % lauseke
+	yhteen	lauseke + lauseke
-	vähennys	lauseke - lauseke
<<	biteittäin siirto vasemmalle	lauseke << lauseke
>>	biteittäin siirto oikealle	lauseke >> lauseke
<	pienempi kuin	lauseke < lauseke
<=	pienempi tai yhtäsuuri	lauseke <= lauseke
>	suurempi kuin	lauseke > lauseke
>=	suurempi tai yhtäsuuri	lauseke >= lauseke
==	yhtäsuuri	lauseke == lauseke
!=	erisuuri	lauseke != lauseke
&	biteittäin JA	lauseke & lauseke
^	biteittäin POIS SULKEVA	lauseke ^ lauseke
|	biteittäin TAI	lauseke | lauseke
&&	looginen JA	lauseke && lauseke
||	looginen TAI	lauseke || lauseke
?:	ehdollinen lauseke	lauseke ? lauseke : lauseke
=	sijoitus	lvalue = lauseke
=,*=,/=,	yhdistetty sijoitus	lvalue += lauseke jne.
%=,+=,-=,<<=, >>=,&=,|=,^=		
throw	heitä poikkeus	throw lauseke
,	pilkku	lauseke , lauseke

4.14 Tyyppikonversiot

Mietitäänpä seuraavaa sijoitusta:

```
int ival = 0;

// kääntäjä antaa tyypillisesti varoituksen
ival = 3.541 + 3;
```

Lopullinen tulos on, että ival:iin sijoitetaan arvo 6. Todelliset vaiheet tuloksen saamiseksi ovat seuraavassa. Olemme lisäämässä kahden eri tyypin arvoja: 3.541 on double-tyyppinen literaali ja 3 on int-tyyppinen literaali. Sen sijaan, että yritettäisiin laskea yhteen kahden eri tyypin arvoja, C++:ssa on joukko *aritmeettisia konversioita*, joilla muunnetaan operandeja yhteiseksi tyypiksi ennen aritmeettisia toimenpiteitä. Sääntönä on aina ylentää pienempi tyyppi suuremmaksi tyypiksi, sillä tarkoituksena estää tarkkuuden menetystä. Tässä tapauksessa kokonaislukuarvo 3 ylennetään double-tyypiksi ennen yhteenlaskun tekemistä. Nämä konversiot suoritetaan kääntäjän toimesta automaattisesti ilman ohjelmoijan väliintuloa (ja usein ohjelmoijan tietämättä). Tästä syystä niitä kutsutaan *implisiittisiksi tyyppikonversioiksi*.

Yhteenlasku ja tulos ovat molemmat double-tyyppiä. Arvo on 6.541. Seuraava vaihe on sijoittaa tulos ival-muuttujaan. Jos sijoituksen vasen ja oikea puoli eivät ole samaa tyyppiä, oikea puoli, jos mahdollista, konvertoidaan vasemman puolen tyypiksi. Tässä tapauksessa ival on tyyppiä int. Konversio double-tyypistä int-tyypiksi tapahtuu automaattisesti katkaisemalla eikä pyöristämällä; desimaaliosa yksinkertaisesti hävitetään. Arvosta 6.541 tulee 6, joka sijoitetaan ival-muuttujaan. Koska konversio double-tyypistä int-tyypiksi saattaa johtaa tarkkuuden menetykseen, useimmat kääntäjät antavat varoituksen.

Koska tyyppikonversio double-tyypistä int-tyypiksi ei tue pyöristystä, se pitää koodata itse. Esimerkiksi:

```
double dval = 8.6;
int ival = 5;

ival += dval + 0.5; // varmista pyöristys
```

Jos haluamme, voimme vähentää aritmeettisia vakiokonversioita määrittämällä *eksplisiittisen tyyppikonversion*:

```
// ohjaa kääntäjää konvertoimaan double-arvo int-tyyppiseksi
ival = static_cast< int >( 3.541 ) + 3;
```

Tässä esimerkissä ohjaamme eksplisiittisesti kääntäjää konvertoimaan double-tyypin arvon int-tyypiksi sen sijaan, että noudattaisimme C++:n aritmeettisia vakiokonversioita.

Tässä kohdassa katsomme yksityiskohtaisesti implisiittistä tyyppikonversiota (tapahtuu automaattisesti kääntäjän toimesta ilman ohjelmoijan väliintuloa kuten ensimmäisessä esimerkissä edellä) ja eksplisiittistä tyyppikonversiota (ohjelmoija ohjaa kääntäjää konvertoimaan olemassaolevan tyypin tiettyyn toiseen tyyppiin kuten toisessa esimerkissä edellä).

4.14.1 Implisiittiset tyyppikonversiot

Kieli määrittelee sisäisten oliotyyppien välille joukon vakiokonversioita, joita kääntäjä käyttää implisiittisesti tarvittaessa. Implisiittiset tyyppikonversiot tapahtuvat seuraavissa yleisissä ohjelmatilanteissa:

- Aritmeettisessa lausekkeessa, jossa on eri tyyppejä. Tässä tapauksessa leveimmästä mukana olevasta tietotyypistä tulee konversion kohdetyyppi. Näitä kutsutaan *aritmeettisiksi konversioiksi*. Esimerkiksi:

```
int    ival = 3;
double dval = 3.14159;

// ival ylennetään double-tyypiksi: 3.0
ival + dval;
```

- Lausekkeessa, jossa sijoitetaan yksi tyyppi toisentyyppiseen olioon. Tässä tapauksessa konversion kohdetyyppi on sen olion tyyppi, johon sijoitetaan. Esimerkiksi ensimmäisessä sijoituksessa literaali 0, joka on tyyppiä int, konvertoidaan osoitintyypiksi int*, joka edustaa osoitteen null-arvoa. Toisessa sijoituksessa double-tyypin arvo katkaistaan int-tyyppiseksi arvoksi.

```
// 0 konvertoidaan osoittimen null-arvoksi, joka on tyyppiä int*
int *pi = 0;

// dval katkaistaan int-tyypiksi: 3
ival = dval;
```

- Funktion käynnistyksessä välitettävässä lausekkeessa, jossa lausekkeen tyyppi poikkeaa muodollisen parametrin tyypistä. Tässä tapauksessa konversion kohdetyyppi on parametrin tyyppi. Esimerkiksi:

```
extern double sqrt( double );

// 2 ylennetään double-tyypiksi: 2.0
cout << "2:n neliöjuuri on "
     << sqrt( 2 ) << endl;
```

- Palautettaessa funktiosta lauseketta, jonka tyyppi poikkeaa palautustyypistä. Tässä tapauksessa konversion kohdetyypi on funktion palautustyyppi. Esimerkiksi:

```
double difference( int ival1, int ival2 )
{
    // paluuarvo ylennetään double-tyypiksi
    return ival1 - ival2;
}
```

4.14.2 Aritmeettiset konversiot

Aritmeettiset konversiot takaavat, että binäärioperaattorin kuten yhteenlaskun tai kertolaskun kaksi operandia ylennetään yhteiseksi tyypiksi, joka sitten edustaa tulostyyppiä. Kaksi yleistä ohjetta ovat seuraavat:

1. Jos on tarpeen, tyypit aina ylennetään leveämmiksi tyypeiksi, jotta estettäisiin tarkkuuden menetyksiä.
2. Kaikissa aritmeettisissa operaatioissa, joihin kuuluu kokonaistyyppejä, jotka ovat pienempiä kuin kokonaisluku, ylennetään ne kokonaisluvuiksi ennen operaation suorittamista.

Säännöt on määritelty tyyppikonversioiden hierarkiaksi kuten seuraavassa. (Aloitamme leveimmällä eli long double -tyypillä.)

Jos yksi operandeista on long double-tyyppiä, silloin toinen konvertoidaan long double -tyypiksi huolimatta siitä, mitä tyyppiä se on. Esimerkiksi seuraavassa lausekkeessa merkkivakio, pieni 'a'-kirjain, ylennetään long double -tyypiksi (sen ASCII-arvo on 97) ja sitten lisätään long double -literaaliin.

```
3.14159L + 'a';
```

Muussa tapauksessa, jos kumpikaan ei ole long double -tyyppiä ja yksi tyypeistä on double-tyyppiä, toinen konvertoidaan double-tyypiksi. Esimerkiksi:

```
int    ival;
float  fval
double dval;

// fval ja ival konvertoidaan double-tyypeiksi ennen yhteenlaskua
dval + fval + ival;
```

Samalla tavalla, jos kumpikaan ei ole double-tyyppiä ja jos yksi tyypeistä on float-tyyppi, silloin toinen konvertoidaan float-tyypiksi. Esimerkiksi:

```
char cval;
int ival;
float fval;

// ival ja cval konvertoidaan float-tyypeiksi ennen yhteenlaskua
cval + fval + ival;
```

Muussa tapauksessa, koska kumpikaan operandeista ei ole kumpaakaan liukulukutyyppiä, niiden pitää molempien olla jotain kokonaistyyppiä. Ennen yhteisen kohdetyypin päättämistä käytetään prosessia nimeltään *kokonaisylennys* (*integral promotion*) kaikille kokonaistyypeille, jotka ovat pienempiä kuin int.

Kokonaisylennyksessä tyypit char, signed char, unsigned char ja short int ylennetään int-tyypiksi. unsigned short int konvertoidaan int-tyypiksi, jos koneen int-tyyppi on tarpeeksi iso esittämään

kaikki unsigned short -tyypin arvot (tämä tapahtuu useimmiten silloin, kun short esitetään puolisanana ja int sanana); muussa tapauksessa se ylennetään unsigned int -tyypiksi.

Tyypit wchar_t ja lueteltu tyyppi ylennetään pienimmäksi kokonaislukutyypiksi, jolla voidaan esittää sen *perustana olevan tyypin* arvot. Jos esimerkiksi on annettu seuraava lueteltu joukko

```
enum status { bad, ok };
```

siihen liittyvät arvot ovat 0 ja 1. Molemmat näistä arvoista voidaan, mutta ei tarvitse, tallentaa char-esitystapaan. Kun arvot todellisuudessa tallennetaan char-tyyppeinä, char edustaa luetellun joukon perustana olevaa tyyppiä. Luetellun joukon, status, kokonaisylennys tapahtuu siis int-tyypiksi ja perustuu sen taustalla olevaan tyyppiin.

Ennen kuin seuraavissa lausekkeissa

```
char cval;
bool found;
enum mumble { m1, m2, m3 } mval;

unsigned long ulong;

cval + ulong; ulong + found; mval + ulong;
```

päätetään yhteinen tyyppi, johon molemmat operandit ylennetään, ylennetään cval, found ja mval int-tyypiksi.

Kun kokonaisylennykset on suoritettu, tyyppien vertailu alkaa jälleen. Jos yksi operandeista on tyyppiä unsigned long, silloin toinen konvertoidaan unsigned long -tyypiksi. Esimerkissämme kaikki kolme oliota, jotka lasketaan yhteen ulong-olion kanssa, ylennetään unsigned long -tyypiksi.

Muussa tapauksessa, jos kumpikaan ei ole unsigned long -tyyppiä ja jos toinen operandi on long-tyyppi, toinen konvertoidaan long-tyypiksi. Esimerkiksi:

```
char cval;
long lval;

// cval ja 1024 ylennetään long-tyypiksi ennen yhteenlaskua
cval + 1024 + lval;
```

On olemassa yksi poikkeus yleiseen long-tyypin konversioon: jos yksi operandeista on long-tyyppi ja toinen on unsigned int -tyyppi, silloin unsigned int ylennetään long-tyypiksi ainoastaan, jos koneen long-tyyppi on tarpeeksi iso esittämään kaikki unsigned int -tyypin arvot (tämä ei päde 32-bittisissä käyttöjärjestelmissä, joissa long ja int molemmat esitetään sanan (*word*) kokoisina); muussa tapauksessa molemmat ylennetään unsigned long -tyypiksi.

Muussa tapauksessa, jos kumpikaan ei ole long-tyyppi ja jos toinen operandeista on unsigned int -tyyppi, toinen konvertoidaan unsigned int -tyypiksi. Muussa tapauksessa molempien operandien täytyy olla int-tyyppisiä.

Vaikka tämä aritmeettisten konversioiden esitys on saattanut olla sinulle enemmän hämmentävää kuin mieltä ylentävää, yleisajatus on säilyttää arvojen tarkkuus lausekkeissa, joissa

on useita eri tyyppejä. Tämä saavutetaan ylentämällä poikkeavat tyypit leveimmän mukana olevan tyypin mukaisiksi.

4.14.3 Eksplisiittiset konversiot

Eksplisiittistä (pakotettua) konversiota kutsutaan *tyyppikonversioksi* (*cast*) ja sitä tukevat seuraavat nimetyt tyyppikonversio-operaattorit: static_cast, dynamic_cast, const_cast ja reinterpret_cast. Vaikka tyyppikonversiot ovat välttämättömiä silloin tällöin, ne ovat myös potentiaalisia ohjelmavirheiden lähteitä: niitä käyttämällä ohjelmoija ottaa pois päältä tai vaimentaa kielen tyyppitarkistuspiirteen. Ennen kuin katsomme, kuinka voimme konvertoida arvoja tyypistä toiseen, paneudumme aluksi siihen, milloin niitä tulisi tehdä.

Osoitin, joka on minkä tahansa muun tyyppinen kuin const, voidaan sijoittaa osoitintyyppiin void*. Osoitinta void* käytetään aina, kun olion täsmällistä tyyppiä ei joko tiedetä tai vaihtelee tilanteiden mukaisesti. Osoitinta void* kutsutaan joskus *geneeriseksi* osoittimeksi, koska se kykenee osoittamaan minkä tahansa tyyppisiä olioita. Esimerkiksi:

```
int ival;
int *pi = 0;
char *pc = 0;
void *pv;

pv = pi; // ok: implisiittinen konversio
pv = pc; // ok: implisiittinen konversio

const int *pci = &ival;
pv = pci; // virhe: pv ei ole const void*;
const void *pcv = pci; // ok
```

void*-osoittimella ei kuitenkaan voida viitata käänteisesti suoraan. Kääntäjällä ei ole tyyppitietoa käytettävissä, joka ohjaisi perustana olevan bittimallin tulkitsemista. Sen sijaan void*-osoitin pitää ensiksi konvertoida tietyn tyyppiseksi osoittimeksi. C++:ssa ei kuitenkaan ole automaattista konversiota void*-osoittimesta tietyn tyyppiseksi osoittimeksi. Mietitäänpä esimerkiksi seuraavaa:

```
#include <cstring>

int ival = 1024;
void *pv;
int *pi = &ival;
const char *pc = "a casting call";

void mumble()
{
    pv = pi; // ok: pv osoittaa ival:ia
    pc = pv; // virhe: ei vakiokonversiota
    char *pstr = new char[ strlen( pc )+1 ];
    strcpy( pstr, pc );
```

```
}
```

Ohjelmoija on selkeästi tehnyt virheen tässä tapauksessa yrittäessään sijoittaa pv:n pc:hen, koska pv osoittaa kokonaislukua eikä merkkitaulukkoa. Kun pc seuraavassa vaiheessa välitetään funktiolle strlen(), joka luulee saavansa null-loppuisen merkkijonon, on ohjelma pahasti virheellinen ja silloin, kun funktio strcpy() suoritetaan, parasta, mitä voimme toivoa, on, että ohjelman suoritus keskeytyy. Se, mikä tekee tästä virheestä vaikean korjata, on sen helppo huomaamattomuus. Tästä syystä C++ vaatii eksplisiittisen tyyppikonversion void*-osoittimelle mihin tahansa eksplisiittiseen tyyppiin:

```
void mumble()
{
        // ok: yhä väärin, mutta nyt se kääntyy!
        // Eksplisiittiset tyyppikonversiot vaativat erityistä huomiota sijoituksissa.
        // Kun ohjelma kaatuu, tyyppikonversioiden tulisi olla ensimmäisiä rakenteita, jotka tutkitaan
        // ensiksi ...

        pc = static_cast< char* >( pv );

        // yhä katastrofi
        char *pstr = new char[ strlen( pc )+1 ];
        strcpy( pstr, pc );
}
```

Toinen syy eksplisiittisten tyyppikonversioiden tekemiseen on tavallisten vakiokonversioiden ohittaminen. Esimerkiksi seuraava yhdistetty sijoitus aluksi ylentää ival:in double:ksi laskeakseen sen yhteen dval:in kanssa ja sitten katkaisee tuloksen int-tyyppiin sijoituksen tehdessään:

```
double dval;
int ival;

ival += dval;
```

Voimme eliminoida tarpeettoman ival:in ylennyksen double:ksi tekemällä dval:ille tyyppikonversion int-tyypiksi:

```
ival += static_cast< int >( dval );
```

Kolmas syy eksplisiittiselle tyyppikonversiolle on hälventää moniselitteisyyttä tilanteessa, jossa yksi tai useampi konversio on mahdollista. Katsomme tätä tapausta lähemmin luvussa 9, kun käsittelemme funktionimien ylikuormitusta.

Eksplisiittisen tyyppikonversion merkintätavan yleinen muoto on seuraava:

```
tyyppikonversion_nimi< tyyppi >( lauseke );
```

Tässä tyyppikonversion_nimi on joku seuraavista: static_cast, const_cast, dynamic_cast tai reinter- pret_cast, tyyppi on konversion kohdetyyppi ja lauseke on arvo, jonka tyyppi konvertoidaan.

Nuo neljä eri tyyppikonversioiden merkintätapaa yrittävät kategorisoida suoritettavia konversioita. const_cast, kuten sen nimi antaa ymmärtää, konvertoi lausekkeestaan vakiotyyppisyy-

den (ja niin myös volatile-olion vaihtelevuuden). Esimerkiksi:

```
extern char *string_copy( char* );
const  char *pc_str;

char *pc = string_copy( const_cast< char* >( pc_str ));
```

Yritys poistaa vakiotyyppisyyttä tyyppikonversiolla käyttämättä jotain kolmesta eri muodosta johtaa käännöksenaikaiseen virheeseen. Samalla tavalla saadaan käännöksenaikainen virhe aikaan, jos yritetään käyttää const_cast-merkintää yleisen tyypin tyyppikonversioon.

Kaikki tyyppikonversiot, jotka kääntäjä tekee implisiittisesti, voidaan tehdä eksplisiittisesti käyttäen static_cast-merkintää. Esimerkiksi:

```
double d = 97.0;
char  ch = static_cast< char >( d );
```

Miksi haluaisimme tehdä tämän? Suuremman aritmeettisen tyypin sijoitus pienempään tyyppiin melkein aina johtaa kääntäjän generoimaan hälytyksen, joka varoittaa meitä mahdollisesta tarkkuuden menetyksestä. Kun teemme eksplisiittisen tyyppikonversion, varoitusilmoitus jätetään antamatta. Eksplisiittinen tyyppikonversio tiedottaa sekä kääntäjälle että ohjelmaa lukevalle, että olemme tietoisia, mutta emme huolissamme mahdollisesta tarkkuuden menetyksestä.

Huonommin käyttäytyviä staattisia tyyppikonversioita — niitä, jotka ovat potentiaalisesti vaarallisia — ovat ne, joissa konvertoidaan void*-osoitintyyppi joksikin eksplisiittiseksi osoitintyypiksi, aritmeettinen arvo luetellun joukon jäseneksi tai kantaluokka johdetuksi luokaksi (tai osoitin tai viittaus sellaisiin luokkiin). (Kantaluokan ja johdetun luokan välisiä konversioita käsitellään luvussa 19.)

Nämä eksplisiittiset tyyppikonversiot ovat potentiaalisesti vaarallisia, koska niiden oikeellisuus riippuu arvosta, joka sattuu olemaan oliossa sillä hetkellä, kun konversio tapahtuu. Jos on annettu esimerkiksi seuraavat esittelyt

```
enum mumble { first = 1, second, third };

extern int ival;
mumble mums_the_word = static_cast< mumble >( ival );
```

konversio ival-tyypistä mumble-tyypiksi on oikein vain, kun ival:in sisältämä arvo on joko 1, 2 tai 3.

reinterpret_cast tekee yleensä matalan tason uudelleentulkinnan operandiensa bittimalleista ja sen oikeellisuus suurelta osin riippuu ohjelmoijan aktiivisesta hallinnasta. Esimerkiksi seuraavassa tyyppikonversiossa

```
complex<double> *pcom;
char *pc = reinterpret_cast< char* >( pcom );
```

ei ohjelmoijan pidä koskaan päästää näkyviltä varsinaista oliota, jota pc osoittaa. Sen välittäminen esimerkiksi string-oliolle, kuten tässä

```
string str( pc );
```

johtaa todennäköisesti str-olion eriskummalliseen käyttäytymiseen suorituksen aikana.

Tämä on hyvä esimerkki siitä, kuinka eksplisiittinen tyyppikonversio voi olla vaarallinen. pc:n alustaminen kompleksiolion osoitteella tapahtuu ilman kääntäjän virhe- tai varoitusilmoitusta ja syynä on eksplisiittinen reinterpret_cast. Kaikissa seuraavissa pc:n käyttötilanteissa sitä kohdellaan char*-oliona ja siten str:n alustus pc:llä on ehdottomasti oikein. Jos kuitenkin kirjoitamme

```
string str( pc );
```

on ohjelman suorituksenaikainen käyttäytyminen tuntematon.

Tämänkaltaisten ongelmien jäljittäminen saattaa osoittautua erityisen vaikeaksi etenkin, jos pcom:in tyyppikonversio pc:ksi tapahtuu eri tiedostossa kuin siellä, mistä str.size() käynnistetään (otaksuttu null-merkki merkkijonon lopusta puuttuu).

Tietyssä mielessä tämä kuvastaa kielen perustavaa laatua olevaa paradoksia. Vahva tyyppitarkistus on tarkoitettu estämään sellaisia virheitä. Eksplisiittinen tyyppikonversio kuitenkin sallii meidän hetkellisesti keskeyttää tyyppitarkistuksen. Siinä vaiheessa, kun alustamme str:n pc:llä, on tyyppitarkistus päällä jälleen; pc näyttää todellakin oikealta tyyppiltä: char*. Mutta sitä se ei tietenkään ole. Eksplisiittisen tyyppikonversion takia kääntäjä ei tiedä sitä.

C++-standardi esitteli nimetyt tyyppikonversio-operaattorit korostaakseen tätä paradoksia tietyllä epäkäytännöllisyydellä, joka ei salli eksplisiittisiä tyyppikonversioita niihin itseensä. Ennen nimettyjä tyyppikonversio-operaattoreita tyyppikonversio tehtiin sulkuparilla (C++-standardi tukee yhtä tätä vanhanaikaista tyyppikonversiota):

```
char *pc = (char*) pcom;
```

Vaikutus on sama, kuin jos käyttäisi merkintää reinterpret_cast, mutta näkyvyys on merkittävästi pienempi, mikä saa tyyppikonversio”veijarin” vaikeammaksi jäljittää.

Kielessä on useita kategorioita eksplisiittisen tyyppikonversion merkintätavasta yhden sijasta kuten esimerkiksi:

```
// ei ole osa C++-kieltä
char *pc = explicit_cast< char* >( pcom );
```

Tuloksena on, että ohjelmoija (kuten myös ohjelmaa lukevat ja sitä käyttävät työkalut) voi selkeästi yksilöidä koodin jokaisen eksplisiittisen tyyppikonversion potentiaalisen riskitason.

Aina, kun olemme kasvotusten ohjelman selittämättömän suorituksenaikaisen käyttäytymisen kanssa, todennäköinen syyllinen on virheellisesti toimiva osoitin. Yksi syy on väärä eksplisiittinen tyyppikonversio; siksi on hyödyllistä käyttää reinterpret_cast-operaattoria kaikkien osoitinten eksplisiittisten tyyppikonversioiden toteuttamiseen ja tunnistamiseen. (Toinen yleinen syy virheelliselle osoittimelle on, että osoitetusta muistialueesta tulee kelpaamaton. Tämä voi tapahtua, kun vahingossa poistamme osoittimen, jota yhä käytetään tai palautamme paikallisen olion osoitteen. Tätä aihetta käsitellään kohdassa 8.4, kun katsomme dynaamista muistinvarausta yksityiskohtaisesti.)

dynamic_cast tukee suorituksenaikaista luokkaolion tunnistusta, jota osoitin tai viittaus osoit-

taa. Sen käsittelyä viivytetään kohtaan 19.1 saakka, jolloin suorituksenaikainen tyypin tunnistus esitellään.

4.14.4 Vanhan tyylin tyyppikonversiot

Juuri esitettyä tyyppikonversion merkintätapaa on kutsuttu tyyppikonversion uuden tyylin merkintätavaksi ja se esiteltiin C++-standardissa. Sitä ennen eksplisiittiset tyyppikonversiot tehtiin huomattavasti yleisemmällä merkintätavalla, jota nyt kutsutaan tyyppikonversion vanhan tyylin merkintätavaksi. Vaikka C++-standardi jatkaa vanhan tyylin merkintätavan tukemista, suosittelemme sen käyttöä vain, kun kirjoitamme koodia, joka tullaan kääntämään joko C-kielen tai C++-esistandardin mukaisesti.

Tyyppikonversion vanhan tyylisellä merkintätavalla on kaksi muotoa:

```
// tyyppikonversion merkintätapa C++-funktiona
tyyppi (lauseke);

// C-kielen tyyppikonversion merkintätapa
(tyyppi) lauseke;
```

Vanhan tyylisiä tyyppikonversioita voidaan käyttää merkintöjen static_cast, const_cast tai reinterpret_cast tilalla C++-standardissa. C++-esistandardissa voidaan käyttää vain vanhan tyylisiä tyyppikonversioita. Koodissa, joka aiotaan kääntää sekä C++- että C-kielessä, voidaan käyttää vain C-kielen tyyppikonversion merkintätapaa.

Tässä on joitakin esimerkkejä tyyppikonversion vanhan tyylisestä merkintätavasta:

```
const char *pc = (const char*) pcom;
int ival = (int) 3.14159;

extern char *rewrite_str( char* );
char *pc2 = rewrite_str( (char*) pc );
int addr_value = int( &ival );
```

Tyyppikonversion vanha merkintätapa säilyttää taaksepäin yhteensopivuuden ohjelmien kanssa, jotka on kirjoitettu C++-esistandardin mukaisesti, ja on yhteensopiva C-kielen kanssa.

Harjoitus 4.21

Olkoon seuraavat määrittelyt:

```
char cval;    int ival;
float fval;   double dval;
unsigned int ui;
```

Yksilöi implisiittisesti tapahtuvat konversiot (vai tapahtuuko niitä yhtään?):

```
(a) cval = 'a' + 3;
(b) fval = ui - ival * 1.0;
(c) dval = ui * fval;
(d) cval = ival + fval + dval;
```

Harjoitus 4.22

Kun on olemassa seuraavat määrittelyt:

```
void *pv;           int ival;
char *pc;           double dval;
const string *ps;
```

kirjoita jokainen seuraavista uudelleen ja käytä nimettyä tyyppikonversiomerkintätapaa:

```
(a) pv = (void*)ps;
(b) ival = int( *pc );
(c) pv = &dval;
(d) pc = (char*) pv;
```

4.15 Esimerkki pinoluokasta

Lisäys- ja vähennysoperaattorien käsittelyn lopuksi esittelemme abstraktion pinosta kuvataksemme näiden operaattoreiden jälki- ja etuliitemuotojen käyttöä. Päätämme tämän luvun käymällä pikaisesti läpi iStack-luokan suunnittelun ja toteutuksen — tarkoittaa pinoa Stack, joka tukee vain int-tyyppisiä elementtejä.

Pino on tietokonetieteessä perustavaa laatua oleva tietoabstraktio, joka mahdollistaa arvojen sisään laittamisen ja takaisin hakemisen LIFO-järjestyksessä — viimeisenä sisään, ensimmäisenä ulos. Pinon kaksi tärkeintä operaatiota ovat uuden arvon *työntö* (*push)* pinoon ja viimeisimmän työnnetyn arvon *veto* (*pop)* pinosta. Muut operaatiot tukevat käyttäjän kyselyjä, kuten onko pino täysi (full()) tai tyhjä (empty()), ja päättelyä pinon koosta (size()) — kuinka monta elementtiä se sisältää. Ensimmäinen toteutuksemme tukee ainoastaan int-tyyppisiä elementtejä. Tässä on sen julkisen rajapinnan esittely:

```
#include <vector>

class iStack {
public:
   iStack( int capacity )
         : _stack( capacity ), _top( 0 ) {}

   bool pop( int &value );
   bool push( int value );

   bool full();
   bool empty();
   void display();

   int size();

private:
   int _top;
   vector< int > _stack;
```

```
};
```

Olemme valinneet kiinteäkokoisen toteutuksen iStack-luokallemme kuvataksemme lisäys- ja vähennysoperaattoreiden jälki- ja etuliitteiden ilmentymien käyttöä (muokkaamme sitä niin, että se kasvaa dynaamisesti luvun 6 loppuun saakka). Tallennamme elementit vektoriin, jonka elementit ovat int-tyyppiä ja nimeltään _stack. _top pitää sisällään seuraavan vapaan paikan, johon arvo voidaan työntää (push()). Arvo _top kertoo pinon nykyisen elementtien lukumäärän. Siitä syystä size()-funktion tarvitsee palauttaa vain _top:

```
inline int iStack::size() { return _top; };
```

empty() palauttaa arvon true, jos _top on yhtä kuin 0 ja full() palauttaa arvon true, jos _top on yhtä kuin _stack.size()-1 (vähennämme arvon 1, koska vektorin elementtien lukumäärä samoin kuin taulukko alkaa nollasta):

```
inline bool iStack::empty() { return _top ? false : true; }
inline bool iStack::full()  {
        return _top < _stack.size()-1 ? false : true;
}
```

Tässä on toteutuksemme funktioista pop() ja push(). Olemme lisänneet tulostusfunktion, jotta voisimme jäljittää jokaisen suoritusta:

```
bool iStack::pop( int &top_value )
{
   if ( empty() )
      return false;

   top_value = _stack[ --_top ];
   cout << "iStack::pop(): " << top_value << endl;

   return true;
}

bool iStack::push( int value )
{

   cout << "iStack::push( " << value << " )\n";

   if ( full() )
      return false;

    _stack[ _top++ ] = value;
   return true;
}
```

Ennen kuin kokeilemme Stack-luokkaamme, lisätkäämme display()-funktio, jonka avulla voimme katsella sisältöjä. Kun pino on tyhjä, se tulostaa seuraavaa:

```
( 0 )
```

Pino, jossa on neljä elementtiä 0, 1, 2 ja 3, generoi

```
( 4 )( bot: 0 1 2 3 :top )
```

Tässä on toteutuksemme:

```
void iStack::display()
{
     cout << "( " << size() << " )( bot: ";

     for ( int ix = 0; ix < _top; ++ix )
               cout << _stack[ ix ] << " ";

 cout << " :top )\n";
}
```

Seuraavassa pienessä ohjelmassa kokeillaan luokkaamme. For-silmukka toistetaan 50 kertaa. Se työntää pinoon jokaisen parillisen arvon: 2, 4, 6, 8 jne. Joka kerta, kun arvo on 5:n kerrannainen eli 5, 10, 15 jne., se näyttää pinon sisällön. Aina, kun arvo on 10:n kerrannainen eli 10, 20, 30 jne., se vetää kaksi viimeistä arvoa pinosta ja näyttää pinon sisällön jälleen.

```
#include <iostream>
#include "iStack.h"

int main() {
   iStack stack( 32 );

   stack.display();
   for ( int ix = 1; ix < 51; ++ix )
   {
      if ( ix%2 == 0 )
          stack.push( ix );

      if ( ix%5 == 0 )
          stack.display();

      if ( ix%10  == 0) {
          int dummy;
          stack.pop( dummy ); stack.pop( dummy );
          stack.display();
      }
   }
}
```

Kun ohjelma käännetään ja suoritetaan, se generoi seuraavan tulostuksen:

```
( 0 )( bot:  :top )
iStack::push( 2 )
iStack::push( 4 )
( 2 )( bot: 2 4  :top )
iStack::push( 6 )
iStack::push( 8 )
iStack::push( 10 )
( 5 )( bot: 2 4 6 8 10  :top )
```

```
iStack::pop(): 10
iStack::pop(): 8
( 3 )( bot: 2 4 6  :top )
iStack::push( 12 )
iStack::push( 14 )
( 5 )( bot: 2 4 6 12 14  :top )
iStack::push( 16 )
iStack::push( 18 )
iStack::push( 20 )
( 8 )( bot: 2 4 6 12 14 16 18 20  :top )
iStack::pop(): 20
iStack::pop(): 18
( 6 )( bot: 2 4 6 12 14 16  :top )
iStack::push( 22 )
iStack::push( 24 )
( 8 )( bot: 2 4 6 12 14 16 22 24  :top )
iStack::push( 26 )
iStack::push( 28 )
iStack::push( 30 )
( 11 )( bot: 2 4 6 12 14 16 22 24 26 28 30  :top )
iStack::pop(): 30
iStack::pop(): 28
( 9 )( bot: 2 4 6 12 14 16 22 24 26  :top )
iStack::push( 32 )
iStack::push( 34 )
( 11 )( bot: 2 4 6 12 14 16 22 24 26 32 34  :top )
iStack::push( 36 )
iStack::push( 38 )
iStack::push( 40 )
( 14 )( bot: 2 4 6 12 14 16 22 24 26 32 34 36 38 40  :top )
iStack::pop(): 40
iStack::pop(): 38
( 12 )( bot: 2 4 6 12 14 16 22 24 26 32 34 36  :top )
iStack::push( 42 )
iStack::push( 44 )
( 14 )( bot: 2 4 6 12 14 16 22 24 26 32 34 36 42 44  :top )
iStack::push( 46 )
iStack::push( 48 )
iStack::push( 50 )
( 17 )( bot: 2 4 6 12 14 16 22 24 26 32 34 36 42 44 46 48 50 :top )
iStack::pop(): 50
iStack::pop(): 48
( 15 )( bot: 2 4 6 12 14 16 22 24 26 32 34 36 42 44 46  :top )
```

Harjoitus 4.23

Jotkut käyttäjistämme ovat pyytäneet peek()-operaatiota. peek() lukee ylimmän arvon poistamatta sitä pinosta, edellyttäen, ettei pino ole tietenkään tyhjä. Toteuta peek() ja lisää se main()-ohjelmaamme kokeillaksesi sitä.

Harjoitus 4.24

Mitkä ovat kaksi pääheikkoutta iStack:in suunnittelussamme? Kuinka ne voitaisiin korjata?