Osa I

C++, yleiskatsaus

Kirjoittamillemme ohjelmille on kaksi päänäkökantaa:

1. Kokoelma *algoritmeja* (tarkoittaa ohjelmoituja käskyjä tietyn tehtävän ratkaisemiseksi).
2. Kokoelma *tietoa*, jota vastaan algoritmit suoritetaan, jotta jokaisesta saataisiin yksilöllinen ratkaisu.

Nämä kaksi ohjelman päänäkökantaa, algoritmit ja tieto, ovat säilyneet muuttumattomina läpi lyhyen tietokonehistorian. Se, mikä on kehittynyt, on niiden väliset suhteet. Tätä suhdetta kutsutaan *ohjelmoinnin paradigmaksi*.

Proseduraalisen ohjelmoinnin paradigmassa ongelma mallinnetaan algoritmijoukolla. Julkisen kirjaston lainaus- ja palautusjärjestelmää materiaaleille kuten kirjat, videot jne esitetään joukkona proseduureja, joista kaksi keskeistä proseduuria ovat kirjaston materiaalien lainaus ja palautus. Tiedot talletetaan erikseen, käsitellään joko yhteisessä paikassa tai välitetään proseduureille. Tunnettuja proseduraalisia kieliä ovat FORTRAN, C ja Pascal. C++ tukee myös proseduraalista ohjelmointia. Yksittäisiä proseduureja kuten check_in(), check_out(), overdue(), fine() jne voidaan pitää funktioina. Osa III, Proseduraalipohjainen ohjelmointi, keskittyy tukeen, jota C++-kielellä on annettavanaan proseduraalisen ohjelmoinnin paradigmalle ja painottaa funktioita, funktiomalleja sekä geneerisiä algoritmeja.

1970-luvulla ohjelmasuunnittelun painopiste siirtyi proseduraalisesta paradigmasta *abstrakteihin tietotyyppeihin* (jota nykyään kutsutaan *oliopohjaiseksi ohjelmoinniksi*). Tässä paradigmassa ongelma mallinnetaan tietoabstraktiojoukolla. C++-kielessä viittaamme näihin abstraktioihin *luokkina* (*classes)*. Esimerkiksi äskeisen kirjaston lainausjärjestelmä esitetään tämän paradigman perusteella vuorovaikutukseksi, joka tapahtuu olioiden ilmentyminä kuten Kirja, Lainaaja, Lainausajan päättyminen (Aika-näkökulma) ja väistämätön Sakkomaksu (Raha-näkökulma), jotka edustavat kirjaston abstraktioita. Algoritmeja, jotka liittyvät jokaiseen luokkaan, kutsutaan luokan *julkiseksi rajapinnaksi* (*public interface*). Tieto on tallennettu yksityisesti jokaiseen olioon ja tiedon käsittely on piilotettu yleisiltä ohjelmilta. Kolme ohjelmointikieltä, jotka tukevat abstraktin tietotyypin paradigmaa, ovat CLU, Ada ja Modula-2. Osa IV, Oliopohjainen ohjelmointi, kuvaa ja käsittelee C++-kielen tukea abstraktin tietotyypin ohjel-

mointiparadigmalle.

Oliokeskeinen ohjelmointi laajentaa abstrakteja tietotyyppejä *periytymisen* (*inheritance*) mekanismin (olemassaolevan toteutuksen "uudelleenkäyttö") ja *dynaamisen sitomisen (dynamic binding)* kautta (olemassaolevan julkisen rajapinnan uudelleenkäyttö). Erityiset tyyppi/alityyppi-suhteet aikaisemmin itsenäisten tyyppien välillä ovat nyt mahdollisia. Kirja, videonauha ja äänite ovat jokainen itsessään *eräänlainen* kirjastomateriaali, vaikka jokaisella on oma lainaus- ja palautuspolitiikkansa. Jaettu julkinen rajapinta ja yksityinen tieto ovat sijoitettuna abstraktiin KirjastoMateriaali-luokkaan. Jokainen tietty kirjastomateriaaliluokka *perii* jaetun käyttäytymisen abstraktilta KirjastoMateriaali-luokalta ja tarvitsee vain ne algoritmit ja tiedot, jotka tukevat sen käyttäytymistä. Kolme tunnetuinta kieltä, jotka tukevat oliokeskeistä paradigmaa, ovat Simula, Smalltalk ja Java. Osa V, Oliokeskeinen ohjelmointi, keskittyy C++-kielen antamaan tukeen oliokeskeiselle ohjelmointiparadigmalle.

C++ on moniparadigmainen kieli. Vaikka ajattelemme sen olevan etupäässä oliokeskeinen kieli, tukee se myös proseduraalista ja oliopohjaista ohjelmointia. Etu on, että voimme tehdä ratkaisun, joka parhaiten sopii ongelmaan — käytännössä mikään paradigmoista ei ole paras ratkaisu jokaiseen ongelmaan. Haittapuolena on, että se saa kielestä suuremman ja monimutkaisemman.

Osassa I esittelemme pikakatsauksen koko C++-kieleen. Eräs syy tähän on, että näin saamme ensin tutustua kielen piirteisiin niin, että voimme vapaammin viitata kielen eri puoliin, ennen kuin käsittelemme ne täydellisesti. Emme esimerkiksi käsittele luokkia yksityiskohtaisesti ennen kuin luvussa 13. Mutta jos odottaisimme siihen saakka, ennen kuin mainitsisimme mitään luokista, päätyisimme esittämään epäedustavia ja melko epäolennaisia ohjelmaesimerkkejä.

Toinen syy kielen laajahkolle ensiesittelylle on esteettinen. Ellet ole joutunut Beethovenin sonaatin kauneuden ja monimutkaisuuden pauloihin tai innostunut Scott Joplinista, on helppo vaihtoehtoisesti tulla kärsimättömäksi ja kyllästyä täysin epäolennaisiin yksityiskohtiin kuten ylennykset, alennukset, oktaavit ja soinnut. Mutta ennen kuin nuo yksityiskohdat hallitaan, musiikin tekeminen jää kaukaiseksi haaveeksi. Sama pätee paljolti ohjelmointiin. Operaattoreiden sidontajärjestyksen sokkeloiden läpikäynti tai aritmetiikan vakiokonversioiden sääntöjen hallinta on välttämätön, mutta vaivalloinen perusta C++-ohjelmoinnin hallitsemiseksi.

Luvussa 1 tutustutaan ensimmäisen kerran kielen peruselementteihin: sisäiset tietotyypit, muuttujat, lausekkeet, lauseet ja funktiot. Luvussa käydään läpi pienin mahdollinen C++-ohjelma, käsitellään ohjelmiemme kääntämisprosessi, käydään lyhyesti läpi esikääntäjä ja katsotaan ensimmäisen kerran syötön ja tulostuksen tukea. Luvussa esitellään useita yksinkertaisia, mutta täydellisiä C++-ohjelmia, jotta lukija rohkaistuisi kääntämään ja suorittamaan niitä.

Luvussa 2 käymme läpi proseduraalisen ohjelman, oliopohjaisen ohjelman ja sitten oliokeskeisen ohjelman toteutuksen taulukosta — tarkoittaa numeroitua kokoelmaa samantyyppisistä elementeistä. Sitten vertaamme taulukkoabstraktiotamme C++-vakiokirjaston vektoriluokkaan ja katsomme ensimmäistä kertaa vakiokirjaston geneerisiä algoritmeja. Matkan aikana motivoimme ja kurkistamme ensimmäisen kerran C++:n tukeen poikkeusten käsittelyssä, malleissa ja nimiavaruuksissa. Itse asiassa koko kieli esitellään, vaikkakin monet yksityiskohdat lykätään tekstin myöhempiin osiin.

Jotkut lukijat saattavat pitää joitakin 2-luvun osia takkuisina läpikäytävinä. Materiaali esitetään tässä ilman täydellistä selostusta, jota kuitenkin normaalisti edellytetään perusteokselta (selostukset tulevat myöhemmissä luvuissa). Jos tunnet, että lukemasi on liian raskasta tai olet kärsimätön yksityiskohtien tasosta, suosittelemme, että vilkuilet tuon osan nopeasti läpi ja palaat siihen myöhemmin, kun materiaali on tutumpaa. Luvussa 3 aloitamme perinteisemmän kertovan tyylin ja niiden lukijoiden, jotka tuntevat luvun 2 epämukavaksi, suositellaan aloittavan sieltä.

1

Alkuun pääsy

Tässä luvussa esitellään kielen perustietotyypit: sisäiset tietotyypit, nimettyjen olioiden määrittely, lausekkeet ja lauseet sekä nimettyjen funktioiden määrittely ja käyttö. Luvussa esitetään pienin mahdollinen C++-ohjelma, käsitellään lyhyesti ohjelmiemme käännösprosessi, käydään läpi esikääntäjä ja katsotaan ensimmäisen kerran syötön ja tulostuksen tukea. Luvussa esitellään useita yksinkertaisia, mutta täydellisiä C++-ohjelmia.

1.1 Ongelman ratkaisu

Ohjelmia kirjoitetaan usein joidenkin ongelmien tai tehtävien ratkaisemiseksi. Katsotaanpa esimerkkiä. Kirjakaupassa kirjoitetaan muistiin kirjan nimi ja julkaisija jokaisesta myydystä kirjasta. Tiedot kirjataan siinä järjestyksessä kuin kirjoja myydään. Joka toinen viikko omistaja laskee käsin jokaisen myydyn kirjan kopioiden lukumäärän ja kuinka monta kirjaa kultakin julkaisijalta on myyty. Lista on aakkosjärjestyksessä julkaisijan mukaan ja sitä käytetään uudelleenjärjestelyyn. Meitä on pyydetty hankkimaan ohjelma tämän työn tekemiseksi.

Eräs menetelmä suuren ongelman ratkaisemiseksi on jakaa se useaan pienempään ongelmaan. Ihannetapauksessa nämä pienemmät ongelmat ovat helpommin ratkaistavissa ja ratkaisevat yhdessä suuremman ongelmamme. Jos vastikään pienempiin osiin jaetut ongelmat ovat yhä liian suuria ratkaistaviksi, jaamme ne puolestaan pienempiin ongelmiin ja jatkamme tätä prosessia niin kauan, että toivottavasti olemme löytäneet ratkaisun jokaiseen osaongelmaan. Tätä strategiaa kutsutaan vaihtelevasti nimillä *hajota ja hallitse* sekä *askel askeleelta jalostaminen*. Kirjakaupan ongelmamme jakautuu helposti neljään aliongelmaan eli tehtävään:

1. Lue myyntitiedot.
2. Laske kirjojen myynnit nimen perusteella ja julkaisijoittain.
3. Lajittele kirjojen nimet julkaisijoittain.
4. Kirjoita tulokset.

Kohdat 1, 2 ja 4 edustavat ongelmia, jotka tiedämme ratkaisevamme, joten niitä ei tarvitse jakaa edelleen pienemmiksi. Kohta 3 on kuitenkin yhä sellainen, josta emme tiedä, kuinka se tehdään. Siispä käytämme menetelmäämme tähän kohtaan:

3a. Lajittele myynnit julkaisijoittain.

3b. Lajittele myynnit kirjan nimen perusteella jokaisen julkaisijan kohdalla.

3c. Vertaa vierekkäisiä kirjan nimiä jokaisen julkaisijaryhmän kohdalla. Kasvata laskuria jokaisen yhtä suuren parin ensimmäisen kohdalla ja poista toinen.

Kohdat 3a, 3b ja 3c edustavat nyt myös ongelmia, jotka tiedämme ratkaisevamme. Koska voimme ratkaista kaikki osoittamamme aliongelmat, olemme itse asiassa ratkaisseet alkuperäisen suuremman ongelman. Lisäksi näemme, että tehtävien alkuperäinen järjestys oli väärä. Vaadittavien toimenpiteiden järjestys on seuraava:

1. Lue myyntitiedot.
2. Lajittele myyntitiedot — ensin julkaisijoittain ja sitten kirjan nimen perusteella julkaisijan sisällä.
3. Yhdistä kirjojen samanlaiset nimet.
4. Kirjoita tulokset uuteen tiedostoon.

Tuloksena olevaa toimenpidesarjaa kutsutaan *algoritmiksi*. Seuraava vaihe on muuntaa algoritmimme tietyksi ohjelmointikieleksi — tässä tapauksessa se on C++.

1.2 C++-ohjelma

C++-kielessä toimenpidettä kutsutaan *lausekkeeksi* (*expression*). Lauseketta, joka päättyy puolipisteeseen, kutsutaan *lauseeksi* (*statement*). C++-ohjelmassa pienin itsenäinen yksikkö on lause. Luonnollisessa kielessä vastaava rakenne on myös lause. Esimerkiksi seuraavat ovat lauseita C++-kielessä:

```
int book_count = 0;
book_count = books_on_shelf + books_on_order;
cout << "the value of book_count: " << book_count;
```

Ensimmäinen on *esittelylause* (*declaration* statement). Ilmaisua book_count kutsutaan vaihtelevasti *tunnukseksi* (*identifier)*, *symboliseksi muuttujaksi* (*symbolic variable)* (tai lyhyesti *muuttujaksi*) tai *olioksi* (*object)*. Se määrittelee alueen tietokoneen muistissa, johon liittyy nimi book_count ja jossa voidaan säilyttää kokonaislukuarvoja. 0 on literaalivakio (*literal constant*). book_count *alustetaan* aluksi arvolla nolla.

Toinen lause on *sijoituslause*. Tietokoneen laskettua yhteen books_on_shelf ja books_on_order, se sijoittaa tuloksen tietokoneen muistialueelle, joka liittyy book_count-muuttujaan. Luultavasti nämä ovat myös kokonaislukumuuttujia, jotka on määritelty ja joihin on sijoitettu arvot ohjelman aikaisemmissa osissa.

Kolmas lause on *tulostuslause*. cout on tulostuskohde, joka liittyy käyttäjän päätteeseen. <<

on tulostusoperaattori. Lause kirjoittaa cout:iin — tarkoittaa käyttäjän päätettä — ensiksi *merkkijonoliteraalin*, joka on lainausmerkkien sisällä, ja sitten arvon, joka on tallennettu tietokoneen muistialueelle muuttujaan nimeltä book_count. Tämän lauseen tulostus näyttää tältä:

```
the value of book_count: 11273
```

edellyttäen, että muuttujan book_count arvo on tässä vaiheessa 11,273.

Lauseet ryhmitetään loogisesti nimettyihin yksikköihin, joita kutsutaan *funktioiksi*. Esimerkiksi kaikki tarpeelliset lauseet myyntitietojen lukemista varten on järjestetty funktioon nimeltään readIn(). Samalla tavalla järjestämme sort()-, compact()- ja print()-funktiot.

C++:ssa täytyy jokaisen ohjelman sisältää funktio nimeltään main(), jonka ohjelmoija tekee, ennen kuin ohjelmaa voidaan ajaa. Seuraavassa näytetään, miten main() voitaisiin määritellä äskeiselle algoritmille:

```
int main()
{
    readIn();
    sort();
    compact();
    print();
    return 0;
}
```

C++-ohjelma aloittaa suorituksensa ensimmäisestä main()-funktion lauseesta. Tässä tapauksessa ohjelma alkaa suorittamalla funktion readIn(). Ohjelma jatkuu suorittaen main()-funktion lauseita peräkkäisessä järjestyksessä. Ohjelma päättyy normaalisti main()-funktion viimeisen lauseen jälkeen.

Funktio muodostuu neljästä osasta: paluutyypistä, funktion nimestä, parametriluettelosta ja funktion rungosta. Kolmea ensimmäistä osaa kutsutaan yhteisesti *funktion prototyypiksi*.

Sulkujen sisällä oleva parametriluettelo sisältää pilkuin eroteltuna yhden tai useampia parametreja; se voi olla myös tyhjä. Funktion runko muodostuu aaltosulkuparista. Se koostuu peräkkäisistä ohjelman lauseista.

Tässä tapauksessa main()-funktion runko *käynnistää* funktiot readIn(), sort(), compact() ja print(). Kun ne on suoritettu, lause

```
return 0;
```

suoritetaan. return, joka on esimääritelty C++-lause, on metodi funktion suorituksen päättämiseksi. Kun sille annetaan arvo, kuten 0, tulee tuosta arvosta funktion *paluuarvo*. Tässä tapauksessa arvo 0 ilmaisee main()-funktion onnistunutta päättymistä. (C++-standardissa main()-funktio palauttaa arvon 0 oletusarvoisesti, ellei eksplisiittistä return-lausetta ole annettu.)

Katsotaanpa nyt, kuinka ohjelma saatetaan valmiiksi suoritusta varten. Aluksi pitää tehdä määrittelyt readIn()-, sort()-, compact()- ja print()-funktioille. Tässä vaiheessa riittävät seuraavat näennäis- eli dummy-määrittelyt:

```
void readIn()  { cout << "readIn()\n";  }
void sort()    { cout << "sort()\n";    }
```

```
void compact() { cout << "compact()\n";  }
void print()   { cout << "print()\n";    }
```

void-määrettä käytetään määriteltäessä funktiota, joka ei palauta arvoa. Kuten on määritelty, jokainen funktio tässä vain yksinkertaisesti ilmoittaa olemassaolostaan käyttäjän päätteelle main()-funktion käynnistämänä. Myöhemmin voimme korvata nämä dummy-funktiot todellisilla funktioilla niin kuin ne toteutetaan.

Tällainen vähitellen kasvattava menetelmä ohjelmien rakentamiseksi on hyödyllistä kontrolloitaessa ohjelmointivirheitä, joita väistämättä teemme. Yritys saada ohjelma toimimaan kokonaan heti on yksinkertaisesti liian monimutkaista ja hämmentävää.

Ohjelman lähdetiedoston nimi muodostuu yleensä kahdesta osasta: tiedostonimestä — esimerkiksi bookstore — ja tiedoston loppuliitteestä. Käytäntö on, että loppuliite palvelee tiedoston sisällön tunnistamista. Tiedosto

```
bookstore.h
```

tulkitaan käytännön mukaisesti *otsikkotiedostoksi* (*header)* sekä C- että C++-kielessä. (C++-standardin otsikkotiedostoilla ei kuitenkaan ole loppuliitettä — ne edustavat kuuluisaa poikkeusta säännöistä.)

Tiedosto

```
bookstore.c
```

tulkitaan käytännön mukaisesti C-ohjelman tekstitiedostoksi, kun taas UNIX-käyttöjärjestelmissä tiedosto

```
bookstore.C
```

tulkitaan käytännön mukaisesti C++-ohjelman tekstitiedostoksi. C++-ohjelmatiedostojen loppuliitteet vaihtelevat eri C++-toteutusten välillä etenkin, koska DOS-ympäristössä ei isoa tai pientä C-kirjainta voida erottaa toisistaan. Muita loppuliitteitä C++-ohjelmien tekstitiedostojen erottamiseksi toisistaan ovat

```
bookstore.cxx
bookstore.cpp
```

Samalla tavalla otsikkotiedostojen loppuliitteet vaihtelevat eri C++-toteutuksissa (tämä on eräs syistä, miksi C++-standardi ei määrittele otsikkotiedoston loppuliitettä). Tarkista kääntäjäsi käyttäjänoppaasta vastaavat loppuliitteet omalta koneeltasi.

Käytä jotain tekstieditoria ja kirjoita seuraava koko ohjelma C++-lähdetiedostoon.

```
#include <iostream>
using namespace std;

void read()    { cout << "read()\n";    }
void sort()    { cout << "sort()\n";    }
void compact() { cout << "compact()\n"; }
void write()   { cout << "write()\n";   }

int main() {
```

```
    read();
    sort();
    compact();
    write();

    return 0;
}
```

iostream on iostream-kirjaston vakio-otsikkotiedosto (huomaa, että sillä ei ole loppuliitettä). Se sisältää tietoa cout-funktiosta, joka on tarpeellinen ohjelmallemme. #include on *esikääntäjän direktiivi*. Se saa aikaan iostream-tiedoston sisällön luvun tekstitiedostoomme. (Kappaleessa 1.3 käsitellään esikääntäjän direktiivejä.)

Nimiä kuten cout, jotka on määritelty C++-vakiokirjastossa, ei voida käyttää ohjelmassamme, ellemme laita esikääntäjän direktiivin

```
#include <iostream>
```

jälkeen lausetta

```
using namespace std;
```

Tätä lausetta kutsutaan *using-direktiiviksi*. C++-vakiokirjaston nimet on esitelty nimiavaruudessa nimeltään namespace std eivätkä ne ole näkyviä ohjelmamme tekstitiedostolle, ellemme tee niitä eksplisiittisesti näkyviksi. Using-direktiivi ohjaa kääntäjää käyttämään kirjastonimiä, jotka on esitelty namespace std:ssä. (Meillä on enemmän sanottavaa nimiavaruuksista ja using-direktiiveistä kappaleissa 2.7 ja 8.5.)[1]

Kun ohjelma on kirjoitettu, sanotaan vaikka prog1.C-tiedostoon, on seuraava vaihe kääntää se. Se tehdään UNIX-käyttöjärjestelmässä seuraavasti ($ edustaa järjestelmän kehotetta):

```
$ CC prog1.C
```

Komennon nimi, jota käytetään C++-kääntäjän käynnistämiseksi, vaihtelee eri toteutuksissa. (Windowsissa komento käynnistetään usein napsauttamalla valikkokohtaa.) C++-kääntäjän komennon nimi on CC käyttämissämme UNIX-työasemissa. Tarkista järjestelmäsi C++-komennon nimi käyttäjän käsikirjasta tai kysy järjestelmänhoitajalta.

Osa kääntäjän työtä on analysoida ohjelmatekstin virheettömyys. Kääntäjä ei havaitse, onko ohjelman tarkoitus oikein, mutta se voi havaita virheitä ohjelman *muodossa*. Ohjelmavirheen kaksi yleisintä muotoa ovat seuraavat:

1. Syntaksivirheet. Ohjelma on tehnyt “kieliopillisen” virheen C++-kielessä. Esimerkiksi:

```
int main ( {   // virhe: puuttuu ')'
   readIn():  // virhe: merkki ':' ei ole sallittu
   sort();
```

1. Tätä kirjoitettaessa kaikki C++-toteutukset eivät tue nimiavaruuksia. Ellei toteutuksesi tue nimiavaruuksia, täytyy using-direktiivi jättää pois. Koska monet tämän kirjan esimerkeistä saatetaan kääntää toteutuksissa, jotka eivät tue nimiavaruuksia, on using-direktiivit jätetty pois useimmista koodiesimerkeistä.

```
    compact();
    print();

    return 0   // virhe: puuttuu ';'
}
```

2. Tyyppivirheet. Jokaiseen tietoalkioon liittyy C++:ssa tyyppi. Esimerkiksi arvo 10 on kokonaisluku. Sana “hei” lainausmerkeissä ilmaisee merkkijonoa. Jos funktiolle, joka olettaa saavansa kokonaislukuargumentin, annettaan merkkijono, kääntäjä ilmoittaa tyyppivirheestä.

Virheilmoitus sisältää rivinumeron ja lyhyen kuvauksen siitä, mitä kääntäjä uskoo meidän tehneen väärin. On hyvä käytäntö korjata virheitä siinä järjestyksessä kuin niistä raportoidaan. Usein yksittäisellä virheellä on sellainen ketjureaktiomainen vaikutus, että kääntäjä raportoi useammista virheistä kuin todellisuudessa on aihetta. Kun virhe on korjattu, voidaan ohjelma kääntää uudelleen. Tätä kiertokulkua kutsutaan usein nimellä *muokkaus-käännös-virheen-etsintä*.

Toinen osa kääntäjän työtä on kääntää muodollisesti oikein oleva ohjelmateksti. Tämä käännös, jota kutsutaan *koodin generoinniksi*, generoi tyypillisesti objekti- eli konekielikäskytekstiä, jota tietokone ymmärtää.

Onnistuneen käännöksen tuloksena on suorituskelpoinen tiedosto. Kun ohjelmamme ajetaan, se generoi seuraavan tulostuksen:

```
readIn()
sort()
compact()
print()
```

C++ määrittelee joukon sisäisiä alkeellisia tietotyyppejä: kokonaisluku ja liukulukujen numeeriset tyypit, merkkityypin ja Boolean-tyypin, joka voi sisältää joko arvon tosi tai epätosi. Jokainen tyyppi liittyy kielen *avainsanoihin*. Jokainen olio ohjelmassamme liittyy tiettyyn tyyppiin. Esimerkiksi

```
int     age = 10;
double price = 19.99;
char   delimiter = ' ';
bool   found = false;
```

määrittelee neljä oliota — `age`, `price`, `delimiter` ja `found` — joiden tyypit ovat kokonaisluku, kaksoistarkkuuden liukuluku, merkki ja Boolean. Jokaiselle tyypille annetaan alkuarvoksi literaalivakio: kokonaisluku 10, liukuluku 19.99, tyhjä merkki ja Boolean-arvo `false`.

Tyyppien *konversiot* tapahtuvat implisiittisesti (automaattisesti) sisäisten tyyppien välillä. Kun esimerkiksi sijoitamme `int`-tyyppiseen `age`-muuttujaan `double`-tyyppisen literaalivakion, kuten tässä:

```
age = 33.333;
```

age-muuttujaan sijoitettu arvo katkaistaan kokonaislukuarvoksi 33. (Näitä *vakiokonversioita* kuten myös tyyppikonversioita käsitellään yksityiskohtaisesti kohdassa 4.14.)

C++-vakiokirjastossa on laajennettu joukko perustietotyyppejä, johon kuuluvat muun muassa merkkijono, kompleksinumero, vektori ja lista. Esimerkiksi:

```
// välttämätön otsikkotiedosto, jos halutaan käyttää merkkijono-oliota
#include <string>
string current_chapter = "Alkuun pääsy";

// välttämätön otsikkotiedosto, jos haluaa käyttää vektorioliota
#include <vector>
vector<string> chapter_titles( 20 );
```

current_chapter on merkkijono-olio, joka alustetaan merkkijonoliteraalilla "Alkuun pääsy". chapter_titles on 20 elementin string-tyyppinen vektori. Tälle ominainen merkintätapa

```
vector<string>
```

ohjaa kääntäjää luomaan vektorityypin, joka pystyy pitämään sisällään merkkijonoelementtejä. Jos määrittelisimme vektoriolion, joka pystyisi pitämään sisällään 20 kokonaislukuelementtiä, kirjoittaisimme

```
vector<int> ivec( 20 );
```

(Meillä on paljon enemmän sanottavaa vektoreista tämän koko tekstin aikana.)

Ei kieli eikä standardikirjasto pysty kumpikaan käytännössä antamaan meille kaikkia niitä tietotyyppejä, joita ohjelmointiympäristömme vaatii. Sen sijaan nykyaikaiset kielet sisältävät tyypin määrittelypiirteen, jonka avulla voimme esitellä uusia tyyppejä kieleen, joita voidaan käyttää enemmän tai vähemmän yhtä helposti kuin sisäisiä tyyppejä. C++:ssa tämä piirre on luokkamekanismi. Vakiokirjaston merkkijono-, vektori- ja listatyypit ovat kaikki luokkia, jotka on ohjelmoitu C++:lla. Niin on myös iostream-kirjasto.

Luokkapiirre on ehkä kaikkein tärkein C++-kielen komponentti ja luvussa 2 teemme mittavan tutustumiskierroksen koko luokkamekanismiin.

1.2.1 Ohjelmankulun kontrollointi

Lauseet suoritetaan oletusarvoisesti suoraviivaisesti peräkkäin. Esimerkiksi aikaisemmassa ohjelmassamme, joka on kirjoitettu uudelleen seuraavassa, read() suoritetaan aina ensiksi, sen jälkeen sort(), compact() ja sitten write().

```
int main()
{
    read();
    sort();
    compact();
    write();

    return 0;
}
```

Jos kuitenkin on ollut erityisen vähän myyntiä, kuten ei yhtään tai yksi kirja, on tuskin kannattavaa lajitella tai yhdistellä sitä, vaikka meidän pitääkin silti tulostaa tuo ainoa tieto tai ilmaista, että myyntiä ei ole ollut. Voimme tehdä tämän ehdollisella *if*-lauseella (tämä edellyttää, että olemme kirjoittaneet uudelleen read()-funktion, jotta se palauttaisi luettujen tietojen lukumäärän):

```
// read() palauttaa luettujen tietojen lukumäärän
// sen paluuarvo on int-tyyppinen
int read() { ... }

// ...

int main()
{
    int count = read();

    // jos tietojen lukumäärä on suurempi kuin 1,
    // suorita silloin sort() ja compact()

    if ( count > 1 ) {
        sort();
        compact();
    }

    if ( count == 0 )
       cout << "ei myyntiä tässä kuussa\n";
    else write();

    return 0;
}
```

Ensimmäisessä if-lauseessa on ehdollinen suoritus, joka perustuu suluissa olevan lausekkeen totuusarvoon. Tässä parannetussa ohjelmassa sort() ja compact() käynnistetään vain, jos count on suurempi kuin 1. Seuraavassa if-lauseessa on suoritukselle kaksi haarautumisvaihtoehtoa. Jos ehto on tosi — tässä tapauksessa, jos count on yhtä kuin 0 — me kirjoitamme yksinkertaisesti, että myyntiä ei ole ollut. Muussa tapauksessa aina, kun count ei ole 0, me käynnistämme write()-funktion. If-lause käsitellään yksityiskohtaisesti kappaleessa 5.3.

Toinen muun kuin peräkkäisen lausesuorituksen muoto on *silmukkalause*. Silmukka toistaa yhtä tai useampaa lausetta niin kauan, kun joku ehto pysyy totena. Esimerkiksi:

```
int main()
{
    int iterations = 0;
    bool continue_loop = true;
```

```
    while ( continue_loop != false )
    {
        iterations++;

        cout << "while-silmukka on suoritettu "
            << iterations << " kertaa\n";

        if ( iterations == 5 )
              continue_loop = false;
    }

    return 0;
}
```

Tässä hieman keksityssä esimerkissä *while*-silmukka suoritetaan viisi kertaa, kunnes iterations on yhtä kuin 5 ja continue_loop-muuttujaan sijoitetaan arvo false. Lause

```
iterations++;
```

kasvattaa iterations-muuttujaa arvolla 1. Käytämme realistisempaa esimerkkiä while-silmukasta kohdassa 1.5 ja katsomme silmukointilauseita tarkemmin luvussa 5.

1.3 Esikääntäjän direktiivit

Esikääntäjän include-direktiivit tekevät otsikkotiedostoista osan ohjelmastamme. Esikääntäjän direktiivit määritetään sijoittamalla #-merkki ohjelmarivin ensimmäiseen sarakkeeseen. Ohjelma, joka käsittelee direktiivejä, on nimeltään *esikääntäjä* (myös *esikäsittelijä* ja on nykyään upotettu itse kääntäjään).

#include-direktiivi lukee sisään nimetyn tiedoston sisällön. Sillä on kaksi eri muotoa:

```
#include <some_file.h>
#include "my_file.h"
```

Jos tiedosto on kulmasulkujen sisällä (<,>), oletetaan sen olevan projekti- tai vakio-otsikkotiedosto. Kun sitä etsitään, tutkitaan ensin esimääritetyt sijaintipaikat, joita voidaan muokata asettamalla polku ympäristömuuttujaan tai komentorivin valitsimella. (Menetelmät tämän tekemiseksi vaihtelevat koneesta toiseen ja suosittelemme, että kysyt kollegaltasi tai katsot käyttäjän käsikirjasta (*User's Guide*) lisätietoja.) Jos tiedoston nimi on lainausmerkeissä, sen oletetaan olevan käyttäjän tekemä otsikkotiedosto. Etsintä alkaa hakemistosta, jossa include-tiedosto sijaitsee.

Include-tiedosto voi itsekin sisältää #include-direktiivin. Koska include-tiedostot voivat sisältää toisia include-tiedostoja, voidaan otsikkotiedosto joskus ottaa mukaan useita kertoja yksittäiseen lähdetiedostoon. Ehdolliset direktiivit vartioivat otsikkotiedostojen moninkertaista mukaanottoa. Esimerkiksi:

```
#ifndef BOOKSTORE_H
#define BOOKSTORE_H
    /* Bookstore.h-tiedoston sisältö tulee tähän */
#endif
```

Ehdollinen direktiivi

```
#ifndef
```

testaa, onko BOOKSTORE_H määritelty aikaisemmin. BOOKSTORE_H on esikääntäjän vakio. (On tapana, että esikääntäjän vakiot kirjoitetaan kaikki isoilla kirjaimilla.) Ellei BOOKSTORE_H:ta ole aikaisemmin määritelty, ehdollisen direktiivin arvoksi tulee tosi ja kaikki seuraavat rivit, jotka tulevat #ifndef-direktiivin jälkeen #endif-direktiiviin saakka, otetaan mukaan ja käsitellään. Vastaavasti, jos #ifndef-direktiivin arvoksi tule epätosi, kaikki rivit sen jälkeen aina #endif-direktiiviin saakka jätetään huomiotta.

Jotta varmistuisimme, että otsikkotiedosto käsitellään vain kerran, laitamme #define-direktiivin

```
#define BOOKSTORE_H
```

heti #ifndef-direktiivin jälkeen. Tällä tavalla BOOKSTORE_H määritellään, kun otsikkotiedoston sisältö käsitellään ensimmäisen kerran ja estää siten #ifndef-direktiivin uudelleentestaamisen myöhemmissä ohjelmatekstin osissa.

Tämä strategia toimii hyvin, edellyttäen, ettei kaksi otsikkotiedostoa, jotka on välttämättä otettava mukaan, testaa esikääntäjän samannimistä vakiota.

#ifdef-direktiiviä käytetään useimmiten ehdolliseen ohjelmakoodin mukaanottamiseen riippuen siitä, onko esikääntäjän vakiota määritelty. Esimerkiksi:

```
int main()
{
#ifdef DEBUG
    cout << "main()-funktion suorituksen aloitus\n";
#endif

    string word;
    vector< string > text;

    while ( cin >> word )
    {
#ifdef DEBUG
        cout << "luettu sana: " << word << "\n";
#endif
        text.push_back( word );
    }

    // ...
}
```

Ellei tässä esimerkissä DEBUG:ia ole määritelty, käännetty ohjelmakoodi on:

```
int main()
{
    string word;
    vector< string > text;

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

    // ...
}
```

Muussa tapauksessa, jos DEBUG on määritelty, kääntäjälle välitetty ohjelmakoodi on:

```
int main()
{
    cout << "main()-funktion suoritus alkaa\n";

    string word;
    vector< string > text;

    while ( cin >> word )
    {
        cout << "luettu sana: " << word << "\n";
        text.push_back( word );
    }

    // ...
}
```

Voimme määritellä esikääntäjän vakion komentoriville, kun käännämme ohjelman käyttäen -D-valitsinta ja siihen liitettyä esikääntäjän vakiota:[2]

```
$ CC -DDEBUG main.C
```

tai käyttäen ohjelman sisällä #define-direktiiviä.

Esikääntäjän nimi, __cplusplus (kaksi alaviivaa), määritellään automaattisesti, kun käännetään C++:lla. Täten voimme ottaa koodia mukaan ehdollisesti sen mukaan, käännämmekö C++:lla vai emme. Esimerkiksi:

```
#ifdef __cplusplus
    // ok: käännämme C++:lla
    // selitämme extern "C" -käsitteen luvussa 7!
    extern "C"
#endif
    int min( int, int );
```

2. Tämä pätee UNIX-järjestelmissä. Windows-ohjelmoijien tulisi tarkistaa asia kääntäjän käyttäjän käsikirjasta.

Nimi __STDC__ määritellään, kun käännetään C-standardissa. Tietystikään __cplusplus- __STDC__-nimiä ei määritellä samaan aikaan.

Kaksi muuta hyödyllistä esimääriteltyä nimeä ovat __LINE__ ja __FILE__. __LINE__ sisältää käännettävän tiedoston nykyisen rivinumeron. __FILE__ sisältää nykyisen käännettävän tiedoston nimen. Niitä voidaan käyttää kuten seuraavassa:

```
if ( element_count == 0 )
    cerr << "Virhe: " << __FILE__
        << " : rivi " << __LINE__
        << "element_count-muuttujan pitää olla nollasta poikkeava.\n";
```

Kaksi esimääriteltyä lisänimeä sisältävät vastaavasti nykyisen käännettävän tiedoston käännösajan (__TIME__) ja päivämäärän (__DATE__). Ajan muoto on hh:mm:ss niin, että esimerkiksi tiedosto, joka on käännetty 17 minuuttia aamukahdeksan jälkeen, esitetään 08:17:05. Jos se olisi käännetty lokakuun 31. päivänä vuonna 1996, joka on torstai, esitettäisiin päivämäärä näin:

```
Oct  31  1996
```

Prosessoitavan tiedoston rivi ja nimi eli vastaavat __LINE__- ja __FILE__-nimien arvot päivitetään. Muut neljä esimääriteltyä nimeä jäävät kuitenkin vakioiksi käännöksen ajaksi. Näitä arvoja ei voi muuttaa.

assert() on yleensä hyödyllinen makro, joka on C-kielen vakiokirjastossa. Käytämme sitä säännöllisesti hyväksi tekstissä lisätessämme välttämättömiä alkuehtoja ohjelmamme oikeaa suoritusta varten. Jos esimerkiksi meidän pitää lukea sisään tekstitiedostoa ja lajitella sanat, niin välttämättömät alkuehdot ovat, että tiedostonimi annetaan meille ja että pystymme sen avaamaan. Jotta voimme käyttää assert()-makroa, täytyy ottaa mukaan siihen liittyvä otsikkotiedosto:

```
#include <assert.h>
```

Tässä on yksinkertainen esimerkki sen käytöstä:

```
assert( filename != 0 );
```

assert()-makromme testaa totuusarvoa, ettei filename ole yhtä kuin 0. Tämä edustaa väitettä (*assertion*) välttämättömästä alkuehdosta ohjelmakoodin oikeaa suoritusta varten, joka seuraa tämän lauseen jälkeen. Jos ehdon arvoksi tulee epätosi — tarkoittaa, että filename on yhtä kuin 0 — väite epäonnistuu: diagnostinen ilmoitus tulostuu ja ohjelma päättyy.

assert.h on C-nimi C-kirjaston otsikkotiedostolle. C++-ohjelma voi viitata C-kirjaston otsikkotiedostoon käyttämällä joko sen C- tai C++-nimeä. Tämän otsikkotiedoston C++-nimi on cassert. C-kirjaston otsikkotiedoston C++-nimi on aina C-nimi, jonka edessä on kirjain c ja josta on tiedoston .h-loppupääte pudotettu pois (kuten aikaisemmin kerrottiin, standardin mukaisissa C++-otsikkotiedostoissa ei ole tiedoston loppupäätettä, koska ne vaihtelevat eri C++-toteutusten välillä).

Esikääntäjän C-otsikkotiedoston #include-direktiivillä ei ole samaa vaikutusta riippuen siitä, käytetäänkö C- vai C++-nimeä. Seuraava esikääntäjän #include-direktiivi

```
#include <cassert>
```

saa aikaan cassert:in sisällön lukemisen tekstitiedostoomme. Mutta koska kaikki C++-kirjaston nimet esitellään nimiavaruudessa std, ei assert()-nimi ole ohjelmamme tekstitiedoston käyttöalueella, ellemme tee sitä eksplisiittisesti näkyväksi seuraavalla using-direktiivillä:

```
using namespace std;
```

Esikääntäjän #include-direktiivillä, joka käyttää C-otsikkotiedostoa

```
#include <assert.h>
```

voidaan assert()-nimeä käyttää suoraan tekstitiedostossa ilman using-direktiiviä[3]. (Kirjastojen valmistajat käyttävät nimiavaruuksia kontrolloidakseen globaalia nimiavaruuden “pilaantumisongelmaa”, jonka heidän kirjastonsa saisi aikaan käyttäjän ohjelman nimiavaruuden kanssa. Kappaleessa 8.5 käsitellään tätä yksityiskohtaisemmin.)

1.4 Muutama sana kommenteista

Kommentit toimivat ihmisten apuna ohjelmiemme lukemista varten. Ne ovat ohjelmoinnin eräs hyvän tavan muoto. Niissä voidaan tehdä yhteenvetoja funktion algoritmista, yksilöidä muuttujan käyttötarkoitusta tai selvittää muulla tavoin epäselvää koodiosaa. Kommentit eivät kasvata suoritettavan ohjelman kokoa. Kääntäjä karsii ne pois ennen koodin generointia.

C++-kielessä on kaksi eri erotinmerkkiä kommentille. Kommenttipari (/*,*/) on sama kuin se, jota käytetään C-kielessä. Kommentin alku ilmaistaan erottimella /*. Kääntäjä käsittää kommenttina kaikki, joka jää /*-erottimen ja vastaavan */-erottimen väliin. Kommenttipari voidaan sijoittaa mihin tahansa sinne, minne voidaan sijoittaa sarkain-, tyhjä- tai rivinvaihtomerkki, ja se voi levitä useammalle ohjelmariville. Esimerkki:

```
/*
 * Tämä on ensimmäinen katsaus C++-luokan määritykseen.
 * Luokkia käytetään sekä oliopohjaisessa että
 * oliokeskeisessä ohjelmoinnissa.  Screen-luokan
 * toteutus on esitetty luvussa 13.
 */
```

3. Tätä kirjoitettaessa eivät kaikki C++-toteutukset tue C-kirjaston otsikkotiedostojen C++-nimiä. Koska monet tämän kirjan esimerkeistä voidaan kääntää toteutuksissa, jotka eivät tue otsikkotiedostojen C++-nimiä, esimerkeissä viitataan C-kirjaston otsikkotiedostoihin joskus C-nimillä ja joskus C++-nimillä.

```
class Screen {
    /* tätä kutsutaan luokan rungoksi */
public:
    void home();   /* siirrä kohdistin kohtaan 0,0 */
    void refresh();/* piirrä Screen uudelleen     */
private:
    /* luokat tukevat “tiedon piilotusta”.  */
    /* tiedon piilotus eristää ohjelman pääsyn */
    /* luokan sisäiseen esitystapaan */
    /*  (sen tietoon). Tämä tehdään  */
    /*  “private:”-otsikon avulla  */
    int height, width;
};
```

Liian monet kommentit ohjelmakoodin sekoitettuna voivat saada koodin epäselväksi. Esimerkiksi width- ja height-muuttujien esittelyä ympäröivät kommentit melkein piilottavat ne. On yleensä suositeltavaa sijoittaa kommenttilohko sen tekstin yläpuolelle, josta se kertoo. Kuten missä tahansa ohjelmiston dokumentaatiossa, pitää kommentit päivittää ohjelmiston kehittyessä. Liian usein kommentit ja koodi, jota ne kommentoivat, ajautuvat erilleen ajan mittaan.

Kommenttipareja ei voi laittaa sisäkkäin — tarkoittaa, että yksi kommenttipari ei voi olla toisen kommenttiparin sisällä. Kokeile kääntää seuraava ohjelma järjestelmässäsi. Se sekoittaa useimmat kääntäjät täysin.

```
#include <iostream>

/*
 * kommenttipareja /* */ ei voi laittaa sisäkkäin.
 * "ei voi laittaa sisäkkäin"-tekstiä pidetään lähdekoodina
 * kuten myös näitä kahta riviä ja seuraavaa.
 */

int main() {
   cout << "hei, maailma\n";
}
```

Eräs tapa korjata sisäkkäisten kommenttiparien ongelma on laittaa tyhjä merkki asteriskin ja vinoviivan väliin:

```
/* * /
```

Asteriski-vinoviiva-paria kohdellaan kommentin erotinmerkkinä vain, jos niiden välissä ei ole tyhjää merkkiä.

Toinen kommentin erotinmerkki ilmaistaan kaksoisvinoviivalla (//), joka toimii yksittäisen rivin kommenttina. Kaikkea ohjelmarivillä erotinmerkistä oikealle kohdellaan kommenttina ja kääntäjä jättää ne huomiotta. Tässä on esimerkiksi Screen-luokka, jossa käytetään kommenttien kaksia eri erotinmerkkejä:

```
/*
 * Tämä on ensimmäinen katsaus C++-luokan määritykseen.
 * Luokkia käytetään sekä oliopohjaisessa että
 * oliokeskeisessä ohjelmoinnissa.  Screen-luokan
 * toteutus on esitetty luvussa 13.
 */

class Screen {
     // Tätä kutsutaan luokan rungoksi
public:
     void home();// siirrä kohdistin kohtaan 0,0
     void refresh();// piirrä Screen uudelleen
private:
    /* Luokat tukevat “tiedon piilotusta”.  */
     /* Tiedon piilotus eristää ohjelman pääsyn */
     /* luokan sisäiseen esitystapaan */
     /*  (sen tietoon). Tämä tehdään  */
     /*  “private:”-otsikon avulla  */

     // Yksityinen tieto tulee tänne ...
};
```

Ohjelmat sisältävät tyypillisesti molempia kommentointimuotoja. Yleensä kommenttiparin sisälle laitetaan useita selitysrivejä. Puolen rivin tai yksittäisen rivin huomautukset ilmaistaan kaksoisvinoviivalla.

1.5 Ensimmäinen katsaus syöttöön ja tulostukseen

C++:ssa syöttö ja tulostus tapahtuu iostream-kirjaston avulla, joka on oliokeskeinen luokkahierarkia, joka on toteutettu C++:aan osana vakiokirjastoa.

Päätteemme syöte, jota kutsutaan *vakiosyötöksi*, on “sidottu” esimääriteltyyn iostream-olioon cin (lausutaan englanniksi “see-in”). Päätteellemme ohjattu tulostus, jota kutsutaan *vakiotulostukseksi*, on sidottu esimääriteltyyn iostream-olioon cout (lausutaan englanniksi “see-out”). Kolmas esimääritelty iostream-olio, cerr (lausutaan englanniksi “see-err”), jota kutsutaan *vakiovirheeksi*, on myös sidottu päätteeseemme. cerr-oliota käytetään tyypillisesti generoitaessa varoitus- ja virheilmoituksia ohjelmiemme käyttäjille.

Kaikkien ohjelmien, jotka aikovat käyttää hyödykseen iostream-kirjastoa, pitää ottaa mukaan siihen liittyvän järjestelmän otsikkotiedosto:

```
#include <iostream>
```

Tulostusoperaattoria (<<) käytetään ohjaamaan arvo vakiotulostukseen tai vakiovirheeseen. Esimerkiksi:

```
int v1, v2;
// ...
cout << "Summa v1 + v2 = ";
cout << v1 + v2;
cout << '\n';
```

Kahden merkin jono, `\n`, edustaa rivinvaihtomerkkiä. Kun rivinvaihtomerkki kirjoitetaan, se päättää rivin ja saa aikaan tulostuksen ohjautumisen seuraavalle riville. Sen sijaan, että kirjoittaisimme eksplisiittisesti rivinvaihtomerkin, voimme käyttää esimääriteltyä iostream-*manipulaattoria* `endl`.

Manipulaattori tekee operaation iostreamille sen sijaan, että käyttäisi tietoa. Esimerkiksi `endl` lisää rivinvaihtomerkin tulostusvirtaan ja tyhjentää sitten tulostuspuskurin. Sen sijaan, että kirjoittaisimme

```
cout << '\n';
```

kirjoitammekin

```
cout << endl;
```

(Tätä esimääriteltyä iostream-manipulaattoria käsitellään luvussa 20.)

Tulostusoperaattorin peräkkäisiä esiintymiä voidaan yhdistää. Esimerkiksi:

```
cout << "Summa v1 + v2 = " << v1 + v2 << endl;
```

Jokainen tulostusoperaattori vuorollaan lähetetään `cout`:iin. Yhdistetty tulostuslause voidaan jakaa usealle riville luettavuuden takia. Seuraavat kolme riviä muodostavat yhden tulostuslauseen:

```
cout << "Summa "
    << v1 << " + "
    << v2 << " = " << v1 + v2 << endl;
```

Samalla tavalla käytetään syöttöoperaattoria (`>>`), kun luetaan arvo vakiosyötteestä. Esimerkiksi:

```
string file_name;
// ...
cout << "Anna avattavan tiedoston nimi: ";
cin >> file_name;
```

Peräkkäisiä syöttöoperaattoreita voidaan myös yhdistellä. Esimerkiksi:

```
string ifile, ofile;
// ...
cout << "Anna syöttö- ja tulostustiedostojen nimet: ";
cin >> ifile >> ofile;
```

Kuinka voisimme lukea tuntemattoman määrän syöttöarvoja? Kappaleen 1.2 lopussa teimme juuri sen. Koodikatkelma

```
string word;
while ( cin >> word )
        // ...
```

lukee yhden merkkijonon standardisyötöstä silmukan joka toistokerralla, kunnes kaikki merkkijonot on luettu. Ehdosta

```
( cin >> word )
```

tulee epätosi, kun tiedoston loppumerkki saavutetaan (se, kuinka tämä tapahtuu, selitetään luvussa 20). Tässä on yksinkertainen esimerkki ohjelmasta, joka käyttää äskeistä koodikatkelmaa:

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

int main()
{
    string word;

    while ( cin >> word )
            cout << "luettu sana on: " << word << '\n';

    cout << "ok: ei enää luettavia sanoja: heippa!\n";
    return 0;
}
```

Seuraavassa on viisi ensimmäistä sanaa James Joycen novellista *Finnegans Wake*:

```
riverrun, past Eve and Adam's
```

Kun nämä sanat kirjoitetaan näppäimistöltä, ohjelman tulostus on seuraava:

```
luettu sana on: riverrun,
luettu sana on: past
luettu sana on: Eve
luettu sana on: and
luettu sana on: Adam's
luettu sana on: ok: ei enää luettavia sanoja: heippa!
```

(Luvussa 6 katsomme, kuinka voimme poistaa välimerkkejä syötön eri merkkijonoista.)

1.5.1 Tiedoston syöttö ja tulostus

Iostream-kirjasto tukee myös tiedoston syöttöä ja tulostusta. Kaikkia operaattoreita, joita voidaan käyttää vakiosyötössä ja -tulostuksessa, voidaan käyttää myös tiedostoihin, jotka avataan joko syöttöä tai tulostusta (tai molempia) varten. Jotta tiedosto voidaan avata joko syöttöä tai tulostusta varten, pitää ottaa mukaan iostream-otsikkotiedoston lisäksi otsikkotiedosto

```
#include <fstream>
```

Kun avaamme tiedoston tulostusta varten, esittelemme oliotyypin ofstream:

```
ofstream outfile( "tiedoston_nimi" );
```

Testataksemme, onnistuiko tiedoston avaus, kirjoitamme

```
// Saa arvon epätosi, jos tiedoston avaus epäonnistui
if ( ! outfile )
      cerr << "Sorry! Emme voineet avata tiedostoa!\n";
```

Samalla tavalla, kun avaamme tiedoston syöttöä varten, esittelemme oliotyypin ifstream:

```
ifstream infile( "tiedoston_nimi" );
if ( ! infile )
      cerr << "Sorry! Emme voineet avata tiedostoa!\n";
```

Tässä on pieni ohjelma, joka lukee syötteenä olevaa tekstitiedostoa nimeltään in_file ja kirjoittaa jokaisen sanan tulostustiedoston nimeltään out_file erotellen ne toisistaan tyhjällä merkillä.

```
#include <iostream>
#include <fstream>
#include <string>
int main()
{
    ofstream outfile( "out_file" );
    ifstream infile( "in_file" );

    if ( ! infile ) {
          cerr << "virhe: syöttötiedostoa ei voi avata!\n";
          return -1;
    }

    if ( ! outfile ) {
          cerr << "virhe: tulostustiedostoa ei voi avata!\n";
          return -2;
  }

  string word;
  while ( infile >> word )
          outfile << word << ' ';

  return 0;
}
```

Luvussa 20 on iostream-kirjaston koko käsittely syöttö ja tulostus mukaan lukien. Nyt, kun meillä on yleiskäsitys siitä, mitä kieli voi tarjota meille, siirrymme esittelemään kielen uusia tietotyyppejä luokka- ja mallipiirteitä käyttäen.