20

Iostream-kirjasto

Syöttö- ja tulostustoimenpiteet tehdään *iostream-kirjaston* kautta, joka on oliokeskeinen luokkahierarkia, jossa käytetään hyväksi sekä moni- että virtuaaliperiytymistä ja joka on tehtynä komponentiksi C++-vakiokirjastoon. Se tukee sisäisten tietotyyppien syöttöä ja tulostusta sekä myös tiedostojen syöttöä ja tulostusta. Lisäksi luokkien suunnittelijat voivat laajentaa iostream-kirjastoa lukemaan ja kirjoittamaan uusia luokkatyyppejä.

Jotta iostream-kirjastoa voitaisiin käyttää ohjelmissamme, pitää ottaa mukaan siihen liittyvä otsikkotiedosto kuten seuraavassa:

```
#include <iostream>
```

Syöttö- ja tulostusoperaatiot on tehty *istream* (syöttövirta)- ja *ostream* (tulostusvirta) -luokkiin (kolmas luokka, *iostream*, on johdettu sekä istream- että ostream-luokista, ja se mahdollistaa kaksisuuntaisen syötön ja tulostuksen). Mukavuudeksemme kirjastossa on määritelty seuraavat vakiovirtaoliot:

1. cin, joka lausutaan englanniksi *see-in*, on istream-luokkaolio ja edustaa *vakiosyöttöä*. Yleensä cin mahdollistaa, että voimme lukea tietoa käyttäjän päätteeltä.
2. cout, joka lausutaan englanniksi *see-out*, on ostream-luokkaolio ja edustaa *vakiotulostusta*. Yleensä cout mahdollistaa, että voimme kirjoittaa tietoa käyttäjän päätteelle.
3. cerr, joka lausutaan englanniksi *see-err*, on ostream-luokkaolio ja edustaa *vakiovirhettä*. cerr on se, jonne ohjaamme ohjelmamme virheilmoitukset.

Tulostus suoritetaan pääosin ylikuormitetulla siirto vasemmalle -operaattorilla (<<). Samalla tavalla syöttö suoritetaan pääosin ylikuormitetulla siirto oikealle -operaattorilla (>>). Esimerkiksi:

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

int main()
{
```

```
    string in_string;

    // kirjoita literaalimerkkijono käyttäjän päätteelle
    cout << "Please enter your name: ";

    // lue käyttäjän syöte in_string-merkkijonoon
    cin >> in_string;

    if ( in_string.empty() )
            // generoi virheilmoitus käyttäjän päätteelle
        cerr << "error: input string is empty!\n";
    else cout << "hello, " << in_string << "!\n";
}
```

Kätevä tapa ajatella näitä kahta operaattoria on, että kumpikin “osoittaa” tiedonsiirtonsa suuntaan. Esimerkiksi

```
>> x
```

siirtää tietoa x:*ään*, kun taas

```
<< x
```

saa tietoa x:*stä*. Kohdassa 20.1 katsotaan tukea, jota iostream-kirjasto tarjoaa tiedon syötölle. Kohdassa 20.5 katsotaan, kuinka voimme laajentaa iostream-kirjastoa niin, että saamme syöttää tietoa uusista luokkatyypeistä. Samalla tavalla kohdassa 20.2 katsotaan tukea, jota iostream-kirjastolla on tarjota tiedon tulostukselle ja kohdassa 20.4 katsotaan, kuinka voimme laajentaa kirjastoa niin, että voimme tulostaa tietoa uusista luokkatyypeistä.

Samalla tavalla kuin iostream-kirjasto tukee lukemista ja kirjoittamista käyttäjän päätteelle, se tukee myös lukemista ja kirjoittamista tiedostoihin. Seuraavat kolme luokkatyyppiä antavat tiedostotukea:

1. ifstream, joka on johdettu istream-luokasta, sitoo tiedoston ohjelman syöttöön.
2. ofstream, joka on johdettu ostream-luokasta, sitoo tiedoston ohjelman tulostukseen.
3. fstream, joka on johdettu iostream-luokasta, sitoo tiedoston ohjelman sekä syöttöön että tulostukseen.

Jotta voisimme käyttää iostream-kirjaston tiedostovirtakomponenttia, pitää ottaa mukaan siihen liittyvä otsikkotiedosto:

```
#include <fstream>
```

(fstream-otsikkotiedostoon sisältyy iostream-otsikkotiedosto, joten molempia ei tarvitse ottaa mukaan.) Samat syöttö- ja tulostusoperaattorit tukevat myös tiedoston syöttöä ja tulostusta. Esimerkiksi:

```
#include <fstream>
#include <string>
#include <vector>
#include <algorithm>
```

```
int main()
{
    string ifile;

    cout << "Please enter file to sort: ";
    cin >> ifile;

    // muodosta ifstream-syöttötiedosto-olio
    ifstream infile( ifile.c_str() );

    if ( ! infile ) {
        cerr << "error: unable to open input file: "
             << ifile << endl;
        return -1;
    }

    string ofile = ifile + ".sort";

    // muodosta ofstream-tulostustiedosto-olio
    ofstream outfile( ofile.c_str() );
    if ( ! outfile) {
        cerr << "error: unable to open output file: "
             << ofile << endl;
        return -2;
    }

    string buffer;
    vector< string, allocator > text;

    int cnt = 1;
    while ( infile >> buffer ) {
            text.push_back( buffer );
            cout << buffer << ( cnt++ % 8 ? " " : "\n" );
    }

    sort( text.begin(), text.end() );

    // ok: tulosta lajitellut sanat tulostustiedostoon
    vector<string,allocator>::iterator iter = text.begin();
    for ( cnt = 1; iter != text.end(); ++iter, ++cnt )
      outfile << *iter
          << (cnt%8 ? " " : "\n" );

    return 0;
}
```

Tässä on malli-istunto ohjelman suorituksesta: meitä pyydetään syöttämään lajiteltava tiedosto. Kirjoitimme alice_emma (syötteemme näkyy lihavoituna mallitulostuksessa). Ohjelma kaiuttaa jokaisen sanan sellaisena kuin se on luettu vakiotulostukseen:

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

Ohjelma kirjoittaa sitten outfile-tiedostoon lajitellut merkkijonot. Välimerkit vaikuttavat tietysti sanojen järjestykseen (korjaamme sen seuraavan kohdan lopussa):

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

Kohdassa 20.6 katsomme tiedostosta syöttöä ja tiedostoon tulostusta tarkemmin.

Iostream-kirjasto tukee myös *muistissa tapahtuvaa syöttöä ja tulostusta* (*in-memory input/output*), jossa virta liitetään ohjelman muistissa olevaan merkkijonoon (string-olio). Siihen voidaan sitten kirjoittaa ja siitä voidaan lukea käyttämällä iostream:in syöttö- ja tulostusoperaattoreita. Voimme määritellä iostream:in string-olion määrittelemällä ilmentymän jostakin seuraavista kolmesta luokkatyypistä:

1. istringstream, joka on johdettu istream-luokasta ja lukee string-merkkijonosta.
2. ostringstream, joka on johdettu ostream-luokasta ja kirjoittaa string-merkkijonoon.
3. stringstream, joka on johdettu iostream-luokasta ja sekä lukee että kirjoittaa string-merkkijonoon.

Jotta näitä luokkia voitaisiin käyttää, pitää ottaa mukaan niihin liittyvä otsikkotiedosto:

```
#include <sstream>
```

(sstream-otsikkotiedosto sisältää iostream-otsikkotiedosto, joten molempia ei tarvitse ottaa mukaan.) Seuraavassa koodikatkelmassa käytetään ostringstream-luokkaa virheilmoituksen muotoiluun. Sitten palautetaan taustalla oleva merkkijono:

```
#include <sstream>

string program_name( "our_program" );
string version( 0.01 );

// ...
```

```
string mumble( int *array, int size )
{
    if ( ! array ) {
        ostringstream out_message;

            out_message << "error: "
                << program_name << "--" << version
                << ": " << __FILE__ << ": " << __LINE__
                << " -- ptr is set to 0; "
                << " must address some array.\n";

            // palauttaa taustalla olevan string-olion
            return out_message.str();
    }
    // ...
}
```

Kohdassa 20.8 katsomme iostream:in string-olioita tarkemmin.

Käytännössä iostream:it tukevat kahta esimääriteltyä merkkityyppiä: char ja wchar_t. Iostream-luokat, jotka olemme kuvanneet tähän saakka (ja joihin keskitymme tämän luvun loppuun saakka) lukevat ja kirjoittavat virtoja, jotka muodostuvat char-tyypistä. Näitä täydentää joukko iostream-olioita, jotka tukevat wchar_t-tyyppiä. Jokainen luokka ja luokkaolio erotetaan niiden char-vastineesta “w”-etuliitteellä. Täten wchar_t-tyyppinen vakiosyöttöolio on nimeltään wcin, vakiotulostus wcout ja vakiovirhe wcerr. Tarvittavat otsikkotiedostot ovat kuitenkin samat sekä char- että wchar_t-virtaluokille ja -luokkaolioille.

wchar_t-syöttö- ja tulostusluokat ovat wostream, wistream ja wiostream. Tiedoston syöttö- ja tulostusluokat ovat wifstream, wofstream ja wfstream. Ostream-string-syöttö- ja tulostusluokat ovat wistringstream, wostringstream ja wstringstream.

20.1 Tulostusoperaattori <<

Yleisimmin käytetty tulostusmetodi on käyttää vasemmalle siirto -operaattoria (<<) cout:iin. Esimerkiksi

```
#include <iostream>

int main() {
   cout << "gossipaceous Anna Livia\n";
}
```

tulostaa seuraavan käyttäjän päätteelle:

```
gossipaceous Anna Livia
```

Tulostusoperaattorit on tehty niin, että ne hyväksyvät minkä tahansa sisäisen tyypin, mukaan lukien const char* -tyypin kuten myös vakiokirjaston string- ja complex-luokkatyypit. Mistä tahansa lausekkeesta, mukaan lukien funktion kutsu, voidaan tehdä argumentti tulostusoperaat-

torille edellyttäen, että lauseke saa arvokseen tietotyypin, jonka tulostusoperaattorin ilmentymä hyväksyy. Esimerkiksi

```
#include <iostream>
#include <string.h>

int main()
{
   cout << "The length of "ulysses" is:\t";
   cout << strlen( "ulysses" );
   cout << '\n';

   cout << "The size of "ulysses" is:\t";
   cout << sizeof( "ulysses" );
   cout << endl;
}
```

tulostaa seuraavaa käyttäjän päätteelle:

```
The length of "ulysses" is:7
The size of "ulysses" is:8
```

(endl on ostream:in *manipulaattori*, joka lisää rivinvaihdon tulostusvirtaan ja tyhjentää sitten ostream-puskurin. Katsomme puskurointia kohdassa 20.9.)

On usein mukavampaa yhdistää tulostusoperaattori yhteen lauseeseen. Esimerkiksi edellinen ohjelma voidaan kirjoittaa uudelleen seuraavasti:

```
#include <iostream>
#include <string.h>

int main()
{
   // tulostusoperaattoreita voidaan yhdistää

   cout << "The length of "ulysses" is:\t"
        << strlen( "ulysses" ) << 'n';

   cout << "The size of "ulysses" is:\t"
        << sizeof( "ulysses" ) << endl;
}
```

Syy siihen, että yhdistämme tulostusoperaattoreita (ja syöttöoperaattoreita myös) on, että lauseke

```
cout << "some string"
```

ratkaistaan vasemman puolen ostream-operaattoriin; tarkoittaa, että lausekkeen tulos on itse cout-olio, jota sitten käytetään seuraavalle tulostusoperaattorille jne. läpi koko sarjan (sanomme, että operaattori << assosioi vasemmalta oikealle).

Sellainen esimääritelty tulostusoperaattori osoitintyypeille on myös olemassa, jolla voidaan tulostaa olion osoite. Oletusarvoisesti nämä arvot tulostetaan heksadesimaalisella ilmaisulla. Esimerkiksi

```
#include <iostream>

int main()
{
   int i = 1024;
   int *pi = &i;

   cout << "i:   " << i
        << "\t&i:\t" << &i << 'n';

   cout << "*pi: " << *pi
        << "\tpi:\t"    << pi  << endl
        << "\t\t&pi:\t" << &pi << endl;
}
```

tulostaa seuraavaa päätteellemme:

```
i:   1024  &i: 0x7ffff0b4
*pi: 1024  pi: 0x7ffff0b4
        &pi: 0x7ffff0b0
```

Näemme myöhemmin, kuinka osoitteita tulostetaan desimaalisella ilmaisulla.

Seuraava ohjelma edustaa eräänlaista arvoitusta. Tarkoituksenamme on tulostaa osoitteen arvo, jonka `pstr` sisältää:

```
#include <iostream>

const char *str = "vermeer";
int main()
{
   const char *pstr = str;
   cout << "The address of pstr is: "
        << pstr << endl;
}
```

Kun ohjelma käännetään ja suoritetaan, tulostus generoi yllättäen seuraavaa:

```
The address of pstr is: vermeer
```

Ongelma on, että tyyppiä `const char*` ei tulkita osoitearvoksi vaan C-tyyliseksi merkkijonoksi. Jotta voisimme tulostaa osoitteen arvon, jonka `pstr` sisältää, pitää kumota `const char*` -tyypin oletuskäsittely. Teemme sen kahdessa vaiheessa: ensiksi teemme tyyppimuunnoksen, jotta saamme `const`:in pois, sitten muunnamme `pstr`:n tyypin `void*`-tyypiksi:

```
<< static_cast<void*>(const_cast<char*>(pstr))
```

Kun ohjelma käännetään ja suoritetaan, odotettu tulostus generoituu:

```
The address of pstr is: 0x116e8
```

Tässä on toinen arvoitus. Tarkoituksenamme on tulostaa kahdesta arvosta suurempi:

```
#include <iostream>

inline void
max_out( int val1, int val2 ) {
   cout << ( val1 > val2 ) ? val1 : val2;
}

int main()
{
   int ix = 10, jx = 20;

   cout << "The larger of " << ix;
   cout << ", " << jx << " is ";

   max_out( ix, jx );

   cout << endl;
}
```

Kun ohjelma käännetään ja suoritetaan, seuraava virheellinen tulostus generoituu:

```
The larger of 10, 20 is 0
```

Ongelma on, että tulostusoperaattorilla on korkeampi sidontajärjestys kuin ehdollisella operaattorilla, ja siitä se syystä tulostaa tosi/epätosi-arvon vertailusta `val1` ja `val2`. Tämä tarkoittaa, että lauseke

```
cout << ( val1 > val2 ) ? val1 : val2;
```

arvioidaan näin

```
(cout << ( val1 > val2 )) ? val1 : val2;
```

Koska `val1` ei ole suurempi kuin `val2`, vertailun tulos on epätosi, joka tulostetaan arvona 0. Jotta operaattorin esimääritelty sidontajärjestys voitaisiin kumota, pitää koko ehdollinen operaattorilauseke laittaa sulkujen sisään:

```
cout << (val1 > val2 ? val1 : val2);
```

Tämä johtaa oikeaan tulostukseen:

```
The larger of 10, 20 is 20
```

Virheellinen tulostus olisi voinut olla paljon helpompi jäljittää, jos literaalit `bool`-arvot, true ja false, olisi tulostettu merkkijonoina arvojen 0 ja 1 sijaan — eli jos tulostuksessa olisi lukenut:

```
The larger of 10, 20 is false
```

Oletusarvoisesti false-literaalin arvo on tulostettuna 0 ja truen 1. Voimme kumota oletusarvon käyttämällä `boolalpha()`-manipulaattoria. Seuraava ohjelma tekee juuri sen:

```
int main()
{
      cout << "default bool values: "
           << true << " " << false
           << "\nalpha bool values: "
           << boolalpha()
           << true << " " << false
           << endl;
}
```

Kun ohjelma suoritetaan, generoituu seuraavaa:

```
default bool values: 1 0
alpha bool values: true false
```

Sisäisellä tyypillä tehdyn taulukon, kuten myös säiliötyyppien kuten vektori tai kartta, tulostus vaatii, että iteroimme läpi ja tulostamme jokaisen yksittäisen elementin. Esimerkiksi:

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

string pooh_pals[] = {
     "Tigger", "Piglet", "Eeyore", "Rabbit"
};

int main()
{
  vector<string> ppals( pooh_pals, pooh_pals+4 );

  vector<string>::iterator iter = ppals.begin();
  vector<string>::iterator iter_end = ppals.end();

  cout << "These are Pooh's pals: ";
  for ( ; iter != iter_end; iter++ )
       cout << *iter << " ";

  cout << endl;
}
```

Sen sijaan, että iteroisimme eksplisiittisesti läpi kaikki säiliön elementit tulostaen jokaisen elementin vuorollaan, voimme käyttää ostream_iterator-iteraattoria ja saamme aikaan saman. Tässä on esimerkkinä yhtäpitävä ohjelma, joka käyttää hyödykseen ostream_iterator-iteraattoria (katso kohdasta 12.4 ostream_iterator-iteraattorin koko käsittely):

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <string>

string pooh_pals[] = {
     "Tigger", "Piglet", "Eeyore", "Rabbit"
```

```
};

int main()
{
   vector<string> ppals( pooh_pals, pooh_pals+4 );

   vector<string>::iterator iter     = ppals.begin();
   vector<string>::iterator iter_end = ppals.end();

   cout << "These are Pooh's pals: ";

   // kopioi jokaisen elementin cout:iin ...
   ostream_iterator< string > output( cout, "" );
   copy( iter, iter_end, output );

   cout << endl;
}
```

Kun ohjelma käännetään ja suoritetaan, generoituu seuraava tulostus:

```
These are Pooh's pals: Tigger Piglet Eeyore Rabbit
```

Harjoitus 20.1

Olkoon seuraavien olioiden määrittelyt:

```
string sa[4] = { "pooh", "tigger", "piglet", "eeyore" };
vector< string > svec( sa, sa+4 );
string robin( "christopher robin" );
const char *pc = robin.c_str();
int ival = 1024;
char blank = ' ';
double dval = 3.14159;
complex purei( 0, 7 );
```

(a) Tulosta jokaisen olion arvo vakiotulostukseen.
(b) Tulosta pc:n osoitteen arvo.
(c) Tulosta ival- ja dval-olion minimiarvo käyttäen ehdollisen operaattorin tulosta:

```
ival < dval ? ival : dval
```

20.2 Syöttö

Syöttöä tukee pääosin siirto oikealle -operaattori (>>). Esimerkiksi seuraavassa ohjelmassa luetaan int-tyyppisiä arvoja vakiosyötöstä ja sijoitetaan vektoriin:

```
#include <iostream>
#include <vector>

int main()
{
```

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

    while ( cin >> ival )
        ivec.push_back( ival );

    // ...
}
```

Alilauseke

```
cin >> ival
```

lukee kokonaisarvon vakiosyötöstä ja, jos se onnistuu, kopioi arvon `ival`:iin. Alilausekkeen tulos on vasemmanpuoleinen istream-olio — tässä tapauksessa `cin`. (Kuten tulemme näkemään, tämä mahdollistaa syöttöoperaattoreiden yhdistelyn.)

Lauseke

```
while ( cin >> ival )
```

lukee arvoja vakiosyötöstä, kunnes `cin` saa arvokseen epätosi. On kaksi yleistä tilannetta, jolloin istream saa arvokseen epätosi: on luettu tiedoston loppumerkki (jolloin olemme lukeneet oikein kaikki tiedoston sisältämät arvot), on kohdattu kelpaamaton arvo kuten 3.14159 (desimaalipiste on kelpaamaton) tai 1e-1 (literaalimerkki `e` on kelpaamaton), eli yleensä mikä tahansa merkkijonoliteraali. Siinä tapauksessa, että luetaan kelpaamaton arvo, istream-olio laitetaan virhetilaan ja kaikki arvojen lukeminen lakkaa. (Kohdassa 20.7 katsomme virhetilanteita tarkemmin.)

On olemassa esimääriteltyjen syöttöoperaattoreiden joukko, joka hyväksyy minkä tahansa tyyppisen sisäisen tietotyypin mukaan lukien C-tyylisen merkkijonon kuten myös vakiokirjaston string- ja complex-luokkatyypit. Esimerkiksi:

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

int main()
{
   int item_number;
   string item_name;
   double item_price;

   cout << "Please enter the item_number, item_name, and price: "
      << endl;

   cin >> item_number;
   cin >> item_name;
   cin >> item_price;

   cout << "The values entered are: item# "
            << item_number << " "
            << item_name << " @$"
```

```
                << item_price << endl;
}
```

Seuraavassa on ohjelmamme mallisuoritus:

```
Please enter the item_number, item_name, and price:
10247 widget 19.99
The values entered are: item# 10247 widget @$19.99
```

Mitä, jos syötämme jokaisen arvon eri riville? Se ei ole ongelma. Syöttöoperaattori hylkää kaikki väliin jäävät tyhjät merkit (välilyönti, sarkain, rivinvaihto, sivunvaihto ja vaunupalautus; katso kohdasta 20.9, kuinka tämä oletuskäyttäytyminen kumotaan):

```
Please enter the item_number, item_name, and price:
10247
widget
19.99
The values entered are: item# 10247 widget @$19.99
```

Arvojen lukeminen johtaa todennäköisemmin iostream-virheeseen kuin arvojen kirjoittaminen. Jos esimerkiksi syötettyjen arvojen järjestys on

```
// virhe: tavaran nimen (item_name) tulisi tulla toisena
BuzzLightyear 10009 8.99
```

johtaa lauseke

```
cin >> item_number;
```

syöttövirheeseen, koska BuzzLightyear ei taatusti ole int-tyyppinen. Kun syöttövirhe tapahtuu, istream-olio saa arvokseen epätosi, jos se testataan. Hieman vahvemmassa toteutuksessa voitaisiin kirjoittaa esimerkiksi:

```
cin >> item_number;
if ( ! cin )
    cerr << "error: invalid item_number type entered!\n";
```

Vaikka lukuoperaatioiden yhdistelyä tuetaan, se ei mahdollista yksittäisten lukuoperaatioiden testausta mahdollista virhetilannetta varten, joten niitä tulisi käyttää vain, kun virheen vaaraa ei ole. Tässä on edellinen ohjelmamme, jossa käytetään lukuoperaatioiden yhdistelyä:

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

int main()
{
   int item_number;
   string item_name;
   double item_price;

   cout << "Please enter the item_number, item_name, and price: "
        << endl;
```

```
    // ok: mutta virhealttiimpi
    cin >> item_number >> item_name >> item_price;

    cout << "The values entered are: item# "
                << item_number << " "
                << item_name << " @$"
                << item_price << endl;
}
```

Merkkijono

```
ab c
d    e
```

muodostuu seuraavista yhdeksästä merkistä: 'a', 'b', ' ' (tyhjä), 'c', '\n' (rivinvaihto), 'd', '\t' (sarkain), 'e' ja '\n'. Seuraava ohjelma lukee kuitenkin vain viisi aakkosmerkkiä syöttöoperaattoria käyttäen:

```
#include <iostream>

int main()
{
    char ch;

    // lue sisään, tulosta sitten jokainen merkki
    while ( cin >> ch )
        cout << ch;
    cout << endl;

    // ...
}
```

Kun ohjelma suoritetaan, on tulostus seuraava:

```
abcde
```

Oletusarvo on, että kaikki tyhjät merkit jätetään huomiotta. Jos haluamme joko lukea tyhjät merkit myös säilyttäen alkuperäisen syötön muodon tai käsitellä tyhjää tilaa (kuten rivinvaihtojen laskenta), on yksi vaihtoehto käyttää hyväksi istream-luokan get()-jäsenfunktiota (ostream-luokan put()-jäsenfunktiota käytetään yleensä get()-jäsenfunktion yhteydessä — katsomme näitä kahta funktiota tarkemmin myöhemmin). Esimerkiksi:

```
#include <iostream>

int main()
{
    char ch;

    // lue jokainen merkki, tyhjät merkit mukaan lukien
    while ( cin.get( ch ))
        cout.put( ch );
    // ...
```

```
}
```

(Toinen vaihtoehto on käyttää noskipws-manipulaattoria.)

Seuraavien kahden merkkijonon katsotaan muodostuvan viidestä merkkijonosta, jotka on eroteltu tyhjällä merkillä toisistaan, jos ne luetaan joko const char* - tai string-syöttöoperaattoreilla:

```
A fine and private place
"A fine and private place"
```

Lainausmerkkien läsnäolo ei saa aikaan, että tyhjät merkit käsiteltäisiin osana laajempaa merkkijonoa. Sen sijaan kahdesta lainausmerkistä tulee ensimmäisen sanan ensimmäinen merkki ja viimeisen sanan viimeinen merkki.

Sen sijaan, että lukisimme jokaisen yksittäisen elementin eksplisiittisesti vuorollaan vakiosyötöstä, voimme käyttää istream_iterator-iteraattoria saman toiminnan aikaansaamiseksi. Esimerkiksi:

```
#include <algorithm>
#include <string>
#include <vector>
#include <iostream>

int main()
{
    istream_iterator< string > in( cin ), eos ;
    vector< string > text ;

    // kopioi vakiosyötöstä luetut arvot text:iin
    copy( in , eos , back_inserter( text ) ) ;

    sort( text.begin() , text.end() ) ;

    // poista kaikki kaksoisarvot
    vector< string >::iterator it ;
    it = unique( text.begin() , text.end() ) ;
    text.erase( it , text.end() ) ;

    // näytä tuloksena oleva vektori
    int line_cnt = 1 ;
    for ( vector< string >::iterator iter = text.begin();
        iter != text.end() ; ++iter , ++line_cnt )
                    cout << *iter
                        << ( line_cnt % 9 ? " " : "\n" ) ;
    cout << endl;
}
```

Ohjelman syöttö on itse ohjelman teksti. Teksti on tallennettu tiedostoon, jolle olemme antaneet nimen istream_iter.C. UNIX-järjestelmässä voimme ohjata tiedoston uudelleen vakiosyöttöön seuraavasti (istream_iter on ohjelman nimi):

```
istream_iter < istream_iter.C
```

(Muissa kuin UNIX-järjestelmissä sinun tulee katsoa *Programmer's Guide* -kirjasta.) Kun ohjelma suoritetaan, generoituu seuraava tulostus:

```
!= " "\n" #include % ( ) *iter ++iter
++line_cnt , 1 9 : ; << <algorithm> <iostream.h>
<string> <vector> = > >::difference_type >::iterator ? allocator back_inserter(
cin copy( cout diff_type eos for in in( int
istream_iterator< it iter line_cnt main() sort( string text text.begin()
text.end() text.erase( typedef unique( vector< { }
```

(Iostream-iteraattoreita käsitellään kohdassa 12.4.)

Esimääriteltyjen syöttöoperaattorien lisäksi voidaan esitellä ylikuormitettuja ilmentymiä syöttöoperaattorista, kun halutaan tukea käyttäjän määrittelemien luokkatyyppien lukemista. Kohdassa 20.5 katsotaan tätä tarkemmin.

20.2.1 Merkkijonosyöttö

Merkkijono voidaan lukea joko C-tyylisenä merkkijonona tai string-luokkatyyppisenä. Suosittelemme, että käytät string-luokkatyyppiä. Pääetu on automaattinen muistinhallinta, joka string-merkkijonotyyppiin liittyy. Jos esimerkiksi haluamme lukea merkkijonon C-tyylisenä merkkitaulukkona, pitää päätellä taulukon koko — tarpeeksi suuri, jotta siihen mahtuu jokainen mahdollinen merkkijono. Tyypillisesti luemme jokaisen merkkijonon tähän taulukkopuskuriin, sitten varaamme sopivasti muistia, jotta merkkijono mahtuu täsmälleen sille varattuun paikkaan keskusmuistiin ja kopioimme merkkijonon puskurista tähän "tilauksen mukaiseen" muistitilaan. Esimerkiksi:

```
#include <iostream>
#include <string.h>

char inBuf[ 1024 ];
try
{
  while ( cin >> inBuf ) {
        char *str = new char[ strlen( inBuf )+1 ];
        strcpy( str, inBuf );
        // ... tee str:lle jotain
        delete [] str;
   }
}
catch( ... ) { delete [] str; throw; }
```

String-tyyppi on huomattavasti yksinkertaisempi hallita:

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

string str;
while ( cin >> str )
    // ... tee jotain string:ille
```

Tämän alikohdan loppuosassa katsomme merkkijonojen lukua käyttäen sekä C-tyylistä merkkitaulukkoa että string-luokan syöttöoperaattoreita. Syöttötekstissä palaamme nuoreen Alice Emmaan:

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

Kirjoitamme sen tekstitiedostoon nimeltään alice_emma ja sitten ohjaamme sen uudelleen ohjelmiemme vakiosyöttöön. Myöhemmin, kun esittelemme tiedostosyöttöä, avaamme sen ja luemme sitä suoraan. Seuraava ohjelma lukee merkkijonoja C-tyylisinä merkkitaulukoina ja päättelee, mikä luetuista merkkijonoista on pisin.

```
#include <iostream>
#include <string.h>

int main()
{
   const int bufSize = 24;
   char buf[ bufSize ], largest[ bufSize ];

   // sisältää tilastoa
   int curLen, max = -1, cnt = 0;
   while ( cin >> buf )
      {
       curLen = strlen( buf );
       ++cnt;

       // uusi pisin sana? tallenna se
       if ( curLen > max ) {
           max = curLen;
           strcpy( largest, buf );
       }
   }

   cout << "The number of words read is "
        << cnt << endl;

   cout << "The longest word has a length of "
```

```
        << max << endl;

    cout << "The longest word is "
        << largest << endl;
}
```

Kun ohjelma käännetään ja suoritetaan, generoituu seuraava tulostus:

```
The number of words read is 65
The longest word has a length of 10
The longest word is creature,"
```

Itse asiassa tulos on virheellinen. beautiful on pisin sana tekstissä ja on yhdeksän merkkiä pitkä. creature tulee kuitenkin valituksi, koska siihen on liittynyt pilkku ja lainausmerkki. Jotta saisimme ohjelmamme tulkitsemaan merkkijonoja niin kuin käyttäjämme odottavat, pitää muut kuin aakkoselliset elementit suodattaa pois.

Ennen kuin teemme sen, katsokaamme kuitenkin ohjelmaa hieman lähemmin. Ohjelmassa jokainen merkkijonon tallennetaan buf-puskuriin, joka on esitelty 24-merkkiseksi taulukoksi. Jos luettaisiin merkkijono, joka on yhtäsuuri tai pitempi kuin 24 merkkiä, buf vuotaisi yli. Ohjelma menisi todennäköisesti virheeseen suorituksen aikana. Istream:in setw()-manipulaattoria voidaan käyttää estämään syöttötaulukon ylivuoto. Esimerkiksi edellistä ohjelmaa voitaisiin muokata seuraavasti:

```
while ( cin >> setw( bufSize ) >> buf )
```

jossa bufSize on buf-merkkitaulukon ulottuvuus. setw() pilkkoo merkkijonon, joka on yhtäsuuri tai suurempi kuin bufSize, kahteen tai useampaan merkkijonoon, joiden maksimipituus on

```
bufSize - 1
```

Null-merkki sijoitetaan jokaisen uuden merkkijonon loppuun. setw():n käyttö vaatii, että ohjelmaan otetaan mukaan iomanip-otsikkotiedosto:

```
#include <iomanip>
```

Elleivät buf-olion näkyvät esittelyt määritä sen ulottuvuutta,

```
char buf[] = "An unrealistic example";
```

voi ohjelma käyttää sizeof-operaattoria — edellyttäen, että tunnus on taulukon nimi, ja viittaus-alueella, joka on näkyvissä lausekkeelle:

```
while ( cin >> setw(sizeof( buf )) >> buf );
```

sizeof-operaattorin käyttö seuraavassa esimerkissä johtaa ohjelman odottamattomaan käyttäytymiseen:

```
#include <iostream>
#include <iomanip>

main()
{
        const int bufSize = 24;
```

```
        char buf[ bufSize ];

        char *pbuf = buf;

        // jokainen merkkijono, joka on pitempi kuin sizeof(char*)
        // pilkotaan kahteen tai useampaan merkkijonoon
        while ( cin >> setw(sizeof(pbuf)) >> pbuf )
            cout << pbuf << endl;
}
```

Kun ohjelma käännetään ja suoritetaan, generoituu seuraavat odottamattomat tulokset:

```
$ a.out
The winter of our discontent

The
win
ter
of
our
dis
con
ten
t
```

Ongelma on, että setw():lle välitetään merkkiosoittimen koko sen merkkitaulukon koon sijasta, johon se osoittaa. Tässä tietyssä koneessa merkkiosoitin on neljä tavua, joten syöttö pilkotaan neljän merkin pituisiksi merkkijonoiksi.

Seuraava yritys korjata erehdys on jopa vielä vakavampi virhe:

```
while ( cin >> setw(sizeof(*pbuf)) >> pbuf )
```

Tarkoitus on välittää setw():lle taulukon koko, johon pbuf osoittaa. Merkintä

```
*pbuf
```

kuitenkin johtaa vain yhteen char-merkkiin. Tässä tapauksessa setw():lle välitetään arvo 1. Jokainen while-silmukan suoritus sijoittaa null-merkin taulukkoon, johon pbuf osoittaa. Vakiovirtaa ei koskaan lueta; silmukka jatkaa suoritustaan, kuten Buzz Lightyear sanoisi, “äärettömyyteen ja sen tuonne puolen”!

String-luokkatyyppiä käyttäen kaikki nämä muistinvarausongelmat poistuvat, koska string-luokka hoitaa ne itse. Tässä on sama ohjelma kirjoitettu käyttäen string-merkkijonoja:

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

int main()
{
    string buf, largest;

    // sisältää tilastoa;
```

```
        int curLen, max = -1, cnt = 0;
        while ( cin >> buf ) {
                curLen = buf.size();
                ++cnt;

                // uusi pisin sana? tallenna se
                if ( curLen > max ) {
                        max = curLen;
                        largest = buf;
                }
        }

        // ... loput samoin kuin ennen
}
```

Tulostus on silti vastoin odotusta ja johtuu siitä, että pilkku ja lainausmerkki tulkitaan osaksi merkkijonoa. Kirjoittakaamme funktio, joka suodattaa nuo elementit merkkijonosta:

```
#include <string>
void filter_string( string &str )
{
    // suodatettavat elementit
    string filt_elems( "\",?.");
    string::size_type pos = 0;

    while (( pos = str.find_first_of( filt_elems, pos ))
            != string::npos )
            str.erase( pos, 1 );
}
```

Tämä toimii hyvin, mutta kovakoodaa joukon elementtejä, jotka haluamme poistaa. Parempi strategia on mahdollistaa, että käyttäjä voi välittää merkkijonon, joka sisältää poistettavat elementit. Jos käyttäjä haluaa käyttää oletuselementtejä, hän voi välittää tyhjän merkkijonon.

```
#include <string>

void filter_string( string &str,
                string filt_elems = string("\",?."))
{
        string::size_type pos = 0;

        while (( pos = str.find_first_of( filt_elems, pos ))
                        != string::npos )
                        str.erase( pos, 1 );
}
```

Seuraava yleisempi versio funktiosta filter_string() hyväksyy iteraattoriparin, joka ilmaisee suodatettavien elementtien alueen:

```
template <class InputIterator>
void filter_string( InputIterator first, InputIterator last,
                string filt_elems = string("\",?."))
{
        for ( ; first != last; first++ )
        {
                string::size_type pos = 0;
                while (( pos = (*first).find_first_of( filt_elems, pos ))
                        != string::npos )
                        (*first).erase( pos, 1 );
        }
}
```

Ohjelmamme, joka käyttää tätä hyödykseen, näyttää tältä:

```
#include <string>
#include <algorithm>
#include <iterator>
#include <vector>
#include <iostream>

bool length_less( string s1, string s2 )
     { return s1.size() < s2.size(); }

int main()
{
   istream_iterator< string > input( cin ), eos;

   vector< string > text;
   // copy on geneerinen algoritmi
   copy(   input, eos, back_inserter( text ));

   string filt_elems( "\",.?;:");
   filter_string( text.begin(), text.end(), filt_elems );

   int cnt = text.size();
   // max_elements on geneerinen algoritmi
   string *max = max_element( text.begin(), text.end(),
                        length_less );
   int len = max->size();

   cout << "The number of words read is "
       << cnt << endl;

   cout << "The longest word has a length of "
       << len << endl;

   cout << "The longest word is "
       << *max << endl;
}
```

Kun käytimme oletusarvoista pienempi kuin -operaattoria max_element()-funktiossa, yllätyimme ohjelman tulostuksesta:

```
The number of words read is 65
The longest word has a length of 4
The longest word is wind
```

wind ei taatusti ole maksimaalinen elementti pituuden ehdoilla. Hetken pähkäilyn jälkeen tajuamme, että string-luokan pienempi kuin -operaattori ei vertaa merkkijonon pituutta, vaan sen aakkosellista suhdetta. Siinä mielessä wind on suurin merkkijono tekstitiedostossamme. Jotta löytäisimme pisimmän merkkijonon, pitää tehdä vaihtoehtoinen pienempi kuin -operaattori: length_less():

```
The number of words read is 65
The longest word has a length of 9
The longest word is beautiful
```

Harjoitus 20.2

Lue vakiosyötöstä seuraavat tyypit: merkkijono, double, merkkijono, int, merkkijono. Tarkista, syntyykö syöttövirhettä.

Harjoitus 20.3

Lue vakiosyötöstä tuntematon määrä merkkijonoja. Tallenna ne listaan. Päättele sekä lyhin että pisin merkkijono.

20.3 Syötön ja tulostuksen lisäoperaattorit

Joissain tilanteissa on välttämätöntä lukea syöttövirtaa tulkitsemattomina tavuina sen sijaan, että luettaisiin tietotyyppeinä kuten char, int, merkkijono jne. Istream-luokan jäsenfunktio get() lukee syöttövirtaa yhden tavun kerrallaan. getline() lukee tavujonon, joka erotetaan joko rivinvaihdolla tai jollain käyttäjän määrittämällä loppumerkillä. get()-jäsenfunktiosta on olemssa kolme muotoa:

1. get(char& ch) lukee yhden merkin syöttövirrasta mukaan lukien tyhjän merkin ja tallentaa sen ch-olioon. Se palauttaa iostream-olion, johon sitä käytetään. Esimerkiksi seuraava ohjelma kerää useita tilastotietoja syöttövirrasta ja kopioi sen sitten tulostusvirtaan samanlaisena:

```
#include <iostream>

int main()
{
    char ch;
    int  tab_cnt = 0, nl_cnt = 0, space_cnt = 0,
         period_cnt = 0, comma_cnt = 0;
```

```
    while ( cin.get( ch )) {
            switch( ch ) {
                    case ' ':  space_cnt++;  break;
                  case '\t': tab_cnt++;    break;
                  case '\n': nl_cnt++;     break;
                  case '.':  period_cnt++; break;
                  case ',':  comma_cnt++;  break;
          }
            cout.put( ch );
    }

    cout << "\nour statistics:\n\t"
       << "spaces: "    << space_cnt  << '\t'
       << "new lines: " << nl_cnt     << '\t'
       << "tabs: "      << tab_cnt    << "\n\t"
       << "periods: "   << period_cnt << '\t'
       << "commas: "    << comma_cnt  << endl;
}
```

Ostream-luokan put()-jäsenfunktio tarjoaa vaihtoehtoisen metodin merkkien tulostamiseen tulostusvirtaan. put() saa char-tyyppisen argumentin ja palauttaa ostream-luokkaolion, jolle se on käynnistetty.

Kun ohjelma käännetään ja suoritetaan, generoituu seuraava tulostus:

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

our statistics:
 spaces: 59      new lines: 6    tabs: 0
 periods: 4      commas: 12
```

2. Toinen get()-versio myös lukee yhden merkin syöttövirrasta. Ero on siinä, että se palauttaa tuon arvon istream-olion sijasta, johon sitä käytetään. Se palauttaa int-tyypin char-tyypin sijasta, koska se palauttaa myös tiedoston loppumerkin, joka usein ilmaistaan arvolla -1 erotukseksi merkistön arvoista. Testataksemme, onko palautettu arvo tiedoston loppumerkki, vertaamme sitä EOF-vakioon, joka on määritelty iostream-otsikkotiedostoon. Muuttuja, johon sijoitetaan get()-funktion paluuarvo, tulisi esitellä int-tyyppiseksi, koska se voi sisältää sekä merkkiarvoja että EOF-merkin. Tässä on yksinkertainen esimerkki:

```
#include <iostream>

int main()
{
```

```
    int ch;

    // vaihtoehtoisesti:
    // while ( ch = cin.get() && ch != EOF )
    while (( ch = cin.get()) != EOF )
        cout.put(ch);

    return 0;
}
```

Käytetäänpä kumpaa tahansa näistä kahdesta get()-funktiosta, vaaditaan seitsemän iteraatiota seuraavien merkkien lukemiseen:

```
a b c
d
```

Luetaan seitsemän merkkiä ('a', tyhjä merkki, 'b', tyhjä merkki, 'c', rivinvaihto, 'd'). Kahdeksas iteraatio kohtaa EOF-merkin. Koska syöttöoperaattori (>>) jättää oletusarvoisesti lukematta tyhjät merkit, luetaan merkit neljällä iteraatiolla, joka palauttaa vuorollaan merkit 'a', 'b', 'c', 'd'. Seuraava get()-funktion muoto voi lukea merkit kahdella iteraatiolla.

3. Kolmannella get()-versiolla on seuraava yleinen tunniste:

```
get(char *sink, streamsize size, char delimiter='\n')
```

sink edustaa merkkitaulukkoa, johon luetut merkit sijoitetaan. size edustaa istream-virrasta luettavien merkkien enimmäismäärää. delimiter ilmaisee merkin, jonka luekemisen jälkeen tulisi lopettaa merkkien lukeminen. Itse erotinta (delimiter) ei lueta, vaan jätetään seuraavaksi merkiksi istream-virtaan. Yleinen virhe on jättää erotinmerkki lukematta ennen seuraavaa get()-funktion lukua. Seuraavassa ohjelmaesimerkissä käytämme istream:in ignore()-jäsenfunktiota, joka heittää erottimet pois. Oletusarvo on, että rivinvaihtomerkki toimii erottimena.

Merkkejä luetaan, kunnes jokin seuraavista tilanteista sattuu. Kaikkien seuraavien tilanteiden jälkeen laitetaan null-merkki taulukon seuraavaan avoimeen positioon.

- size-1 merkkiä on luettu
- tiedoston loppumerkki on kohdattu
- erotinmerkki on kohdattu (Jälleen: tuota merkkiä ei laiteta taulukkoon, vaan jätetään seuraavaksi merkiksi istream:iin.)

get()-funktion paluuarvo on istream-olio, jonka kautta se on käynnistetty. (gcount() antaa luettujen merkkien lukumäärän.) Tässä on yksinkertainen esimerkki sen käytöstä:

```
#include <iostream>

int main()
```

```
{
     const int max_line = 1024;
     char line[ max_line ];

    while ( cin.get( line, max_line ))
    {
                // maksimissaan luetaan: max_line - 1, jotta null mahtuu
                int get_count = cin.gcount();
                cout << "characters actually read: "
                    << get_count << endl;

                // tee riville jotain

                // jos kohtasimme rivinvaihdon,
                   // hylkäämme sen ennen kuin luemme seuraavan rivin
                   if ( get_count < max_line-1 )
                         cin.ignore();
          }
}
```

Kun tätä käytetään nuoren Alice Emman tekstiin, ohjelma generoi seuraavan tulostuksen:

```
characters actually read: 52
characters actually read: 60
characters actually read: 66
characters actually read: 63
characters actually read: 61
characters actually read: 43
```

Jotta voisimme testata sen käyttäytymistä vielä paremmin, teimme pitemmän rivin kuin max_line merkkiä ja sijoitimme sen tiedoston alkuun, joka sisälsi Alice Emman tekstin:

```
characters actually read: 1023
characters actually read: 528
characters actually read: 52
characters actually read: 60
characters actually read: 66
characters actually read: 63
characters actually read: 61
characters actually read: 43
```

Oletusarvo on, että ignore() lukee ja hylkää yhden merkin istream-oliosta, johon sitä käytetään, mutta eksplisiittinen pituus ja erotin voidaan määrittää. Sen yleinen tunniste on seuraava:

```
ignore( streamsize length = 1, int delim = traits::eof )
```

ignore() lukee ja hylkää length merkkiä istream-oliosta tai kaikki merkit erottimeen saakka se mukaan lukien tai kunnes tiedoston loppumerkki kohdataan. Se palauttaa istream-olion, johon sitä käytetään.

Koska ohjelmoijat usein unohtavat hylätä erottimen ennen kuin käyttävät uudelleen get()-funktiota, on parempi käyttää getline()-jäsenfunktiota get()-jäsenfunktion sijasta, koska se hylkää erottimen sen sijaan, että jättäisi sen istream:in seuraavaksi merkiksi. getline()-jäsenfunktion tunniste on samanlainen kuin kolmeargumenttisen get()-funktion muoto (se myös palauttaa istream-olion, jonka kautta se on käynnistetty):

```
getline(char *sink, streamsize size, char delimiter='\n')
```

Koska sekä getline() että kolmeargumenttinen muoto get()-funktiosta saattavat lukea size merkkiä tai vähemmän, on usein välttämätöntä kysellä istream:ilta, kuinka monta merkkiä todellisuudessa luettiin. Istream:in jäsenfunktio gcount() antaa juuri tuon tiedon; se palauttaa merkkien lukumäärän, joka todellisuudessa luettiin viimeisellä get()- tai getline()-kutsulla.

Ostream:in write()-jäsenfunktio tarjoaa vaihtoehtoisen metodin merkkitaulukon tulostukseen. Sen sijaan, että tulostaisi merkkejä, kunnes päättävä null-merkki löytyy, se tulostaa tietyltä pituudelta merkkejä null-merkit mukaan lukien, jos niitä on. Sillä on seuraava funktion tunniste:

```
write( const char *sink, streamsize length )
```

length määrittää näytettävien merkkien lukumäärän. write() palauttaa ostream-luokkaolion, jonka kautta se on käynnistetty.

Ostream:in päinvastainen jäsenfunktio write()-funktiolle on read()-funktio, jonka tunniste on määritelty seuraavasti:

```
read( char* addr, streamsize size )
```

read() lukee size yhtenäistä tavua syöttövirrasta ja sijoittaa ne addr-osoitteen alkuun. gcount() palauttaa viimeisessä read()-funktion käynnistyksessä luettujen tavujen lukumäärän. read() palauttaa istream-luokkaolion, joka käynnistää sen. Tässä on esimerkki getline()-, gcount()- ja write()-funktion käytöstä:

```
#include <iostream>

int main()
{
        const lineSize = 1024;
        int lcnt = 0; // kuinka monta riviä luettu
        int max = -1; // pisimmän rivin koko
        char inBuf[ lineSize ];

        // lukee 1024 merkkiä tai rivinvaihtoon saakka
        while (cin.getline( inBuf, lineSize ))
        {
                // kuinka monta merkkiä todella luettiin
                int readin = cin.gcount();

                // tilastoa: rivilukumäärä, pisin rivi
                ++lcnt;
                if ( readin > max )
```

```
                max = readin;

            cout << "Line #" << lcnt
                << "\tChars read: " << readin << endl;

            cout.write( inBuf, readin).put('\n').put('\n');
        }

        cout << "Total lines read: " << lcnt << endl;
        cout << "Longest line read: " << max << endl;
}
```

Kun ohjelmalla ajetaan muutamia ensimmäisiä rivejä Moby Dickistä, ohjelma generoi seuraavan tulostuksen:

```
Line #1 Chars read: 45
Call me Ishmael.  Some years ago, never mind

Line #2 Chars read: 46
how long precisely, having little or no money

Line #3 Chars read: 48
in my purse, and nothing particular to interest

Line #4 Chars read: 51
me on shore, I thought I would sail about a little

Line #5 Chars read: 47
and see the watery part of the world.  It is a

Line #6 Chars read: 43
way I have of driving off the spleen, and

Line #7 Chars read: 28
regulating the circulation.

Total lines read: 7
Longest line read: 51
```

Istream:in getline()-funktio tukee vain syöttöä merkkitaulukkoon. Vakiokirjastossa on kuitenkin jäsenetön getline()-ilmentymä, joka lukee syöttöä string-olioon. Sen yleinen tunniste on seuraava:

```
getline( istream &is, string str, char delimiter );
```

Tämä getline()-ilmentymä käyttäytyy seuraavasti: merkkejä luetaan enintään str::max_size()-1. Jos syöttö ylittää tämän rajoituksen, lukuoperaatio epäonnistuu ja istream-olio asetetaan virhetilaan. Muussa tapauksessa syöttö loppuu, kun joko erotin luetaan (se hylätään istream:ista, mutta ei lisätä merkkijonoon) tai tiedoston loppumerkki kohdataan.

On olemassa kolme muuta yleisesti kiinnostavaa istream-operaattoria:

```
// työnnä merkki takaisin iostream:iin
putback( char c );

// asettaa osoittimen 'seuraavaan' istream:in jäseneen ja siitä yhden taaksepäin
unget();

// palauttaa seuraavan merkin (tai EOF)
// mutta ei lue sitä
peek();
```

Seuraavassa koodikatkelmassa kuvataan, kuinka joitakin näitä operaattoreita voidaan käyttää:

```
char ch, next, lookahead;

while ( cin.get( ch ))
{
        switch (ch) {
        case '/':
            // onko se rivikommentti? käytä peek()-operaattoria, niin näet:
            // kyllä? käytä ignore():a rivin loppuun saakka
            next = cin.peek();
            if ( next == '/' )
                    cin.ignore( lineSize, '\n' );
            break;
        case '>':
            // etsi >>=
            next = cin.peek();
            if ( next == '>' ) {
                    lookahead = cin.get();
                    next = cin.peek();
                    if ( next != '=' )
                            cin.putback( lookahead );
            }
            // ...
  }
```

Harjoitus 20.4

Lue seuraava merkkijono vakiosyötöstä mukaan lukien kaikki tyhjät merkit ja kaiuta jokainen merkki takaisin vakiotulostukseen:

```
a   b c
d       e
f
```

Harjoitus 20.5

Lue lause “riverrun, from bend of bay to swerve of shore” (a) yhdeksänä merkkijonona ja (b) yhtenä merkkijonona.

Harjoitus 20.6

Käytä getline():ä ja gcount():ia ja lue rivejä vakiosyötöstä. Päättele pisin luettu rivi (varmistu, että rivi, joka vaatii useita getline()-kutsuja, lasketaan yhdeksi riviksi).

20.4 Tulostusoperaattorin << ylikuormitus

Kun toteutamme luokkatyypin, pitää tehdä ylikuormitettu ilmentymä sekä syöttö- että tulostusoperaattorista, jos haluamme luokkamme tukevan noita operaatioita. Tässä kohdassa katsomme, kuinka tulostusoperaattoria ylikuormitetaan. Syöttöoperaattorin ylikuormitus on seuraavan kohdan aiheena. Tässä on ylikuormitettu ilmentymä tulostusoperaattorista WordCount-luokalle:

```
#include <iostream>

class WordCount {
   friend ostream&
        operator<<(ostream&, const WordCount&);
public:
   WordCount( string word, int cnt=1 );
   // ...
private:
        string word;
        int occurs;
};

ostream&
operator <<( ostream& os, const WordCount& wd )
{   // muotoile: <occurs> word
        os << "< " << wd.occurs << " > "
            << wd.word;
        return os;
}
```

Suunnitteluun liittyvä kysymys on, tulisiko luokan tulostusoperaattorin generoida loppuun rivinvaihtoa. Tuloksena on, että sisäisten tyyppien tulostusoperaattorit eivät tee niin, joten käyttäjät yleensä olettavat, että luokkailmentymätkään eivät tee niin. Parempi suunnitteluvalinta luokkailmentymälle on, että tulostusoperaattori ei generoi loppuun rivinvaihtoa.

Toteutuksen jälkeen WordCount:in ilmentymää tulostusoperaattorista voidaan käyttää vapaasti muiden tulostusoperaattorien kanssa. Esimerkiksi

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

int main()
{
   WordCount wd( "sadness", 12 );
   cout << "wd:\n" << wd << endl;
   return 0;
}
```

tulostaa seuraavaa käyttäjän päätteelle:

```
wd:
<12> sadness
```

Tulostusoperaattori on binäärioperaattori (kaksiargumenttinen), joka palauttaa ostream-viittauksen. Ylikuormitetun määrittelyn yleinen runko näyttää seuraavalta:

```
// ylikuormitetun tulostusoperaattorin yleinen runko
ostream&
operator <<( ostream& os, const ClassType &object )
{
   // mikä tahansa erityislogiikka olion valmisteluun

   // jäsenten todellinen tulostus
   os <<  // ...

   // palauta ostream-olio
   return os;
}
```

Sen ensimmäinen argumentti on viittaus ostream-olioon. Toinen on yleensä const-viittaus tiettyyn luokkatyyppiin. Paluutyyppi on ostream-viittaus. Sen arvo on aina ostream-olio, jota vastaan tulostusoperaattoria on käytetty.

Koska ensimmäinen argumentti on ostream-viittaus, pitää tulostusoperaattori määritellä jäsenettömäksi funktioksi. (Kohdassa 15.1 käsitellään tätä tarkemmin.) Kun operaattorin pitää päästä muihin kuin julkisiin jäseniin, se pitää esitellä ystävänä luokalle. (Ystävät on käsitelty kohdassa 15.2.)

Location on luokka, joka sisältää rivin ja sarakkeen jokaisen sanan esiintymästä. Tässä on sen määrittely:

```
#include <iostream>
```

```
class Location {
   friend ostream& operator<<( ostream&, const Location& );
public:
   Location( int line, int col )
         : _line( line ), _col( col ) {}
private:
   short _line;
   short _col;
};

ostream& operator <<( ostream& os, Location& lc )
{
   // Location-olion tulostus:  < 10,37 >
   os << "<" << lc._line
      << "," << lc._col << "> ";

   return os;
}
```

Määritelkäämme uudelleen WordCount niin, että se sisältää sekä vektorin Location-luokkaolioista occurList että string-luokkaolion word:

```
#include <vector>
#include <string>
#include <iostream>
#include "Location.h"

class WordCount {
   friend ostream& operator<<(ostream&, const WordCount&);

public:
        WordCount(){}
        WordCount( const string &word ) : _word(word){}
        WordCount( const string &word, int ln, int col )
                    : _word( word ){ insert_location( ln, col ); }

        string word()   const { return _word; }
        int    occurs() const { return _occurList.size(); }
        void   found( int ln, int col )
                   { insert_location( ln, col ); }

private:
        void insert_location( int ln, int col )
                    { _occurList.push_back( Location( ln, col )); }

        string            _word;
        vector< Location > _occurList;
};
```

Sekä string- että Location-luokka määrittelevät ilmentymän operator<<()-operaattorista. Tässä on uusi määrittely WordCount:in tulostusoperaattorista:

```
ostream&
operator <<( ostream& os, const WordCount& wd )
{
        os  << "<" << wd._occurList.size() << "> "
            << wd._word << endl;

        int cnt = 0, onLine = 6;
        vector< Location >::const_iterator first =
            wd._occurList.begin();

        vector< Location >::const_iterator last =
            wd._occurList.end();

        for ( ; first != last, ++first )
        {
            // os << Location
            os << *first << " ";

            // muotoilua: 6 riville
            if ( ++cnt == onLine )
              { os << "\n"; cnt = 0;  }

        }
        return os;
}
```

Tässä on ohjelma, joka kokeilee uutta WordCount-määrittelyä. Yksinkertaisuuden vuoksi esiintymät on koodattu käsin.

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

int main()
{
  WordCount search( "rosebud" );

  // yksinkertaisuuden vuoksi käsin koodatut esiintymät
  search.found(11,3);  search.found(11,8);
  search.found(14,2);  search.found(34,6);
  search.found(49,7);  search.found(67,5);
  search.found(81,2);  search.found(82,3);
  search.found(91,4);  search.found(97,8);

  cout << "Occurrences: " << "\n"
      << search << endl;

  return 0;
```

```
}
```

Kun ohjelma käännetään ja suoritetaan, generoituu seuraava tulostus:

```
Occurrences:
<10> rosebud
<11,3>  <11,8>  <14,2>  <34,6>  <49,7>  <67,5>
<81,2>  <82,3>  <91,4>  <97,8>
```

Tämän ohjelman tulostus on tallennettu tiedostoon nimeltään output. Seuraava ponnistuksemme on määritellä syöttöoperaattori, jolla luemme sen takaisin sisään.

Harjoitus 20.7

Olkoon seuraava Date-luokan määrittely:

```
class Date {
public:
    // ...
private:
    int month, day, year;
};
```

Tee ylikuormitettu ilmentymä tulostusoperaattorista, joka (a) generoi muodon

```
// kirjoita kuukausi
September 8th, 1997
```

(b) generoi muodon

```
9 / 8 / 97
```

(c) Kumpi on parempi, vai onko kumpikaan? Miksi?
(d) Tulisiko Date:n tulostusoperaattorin olla ystäväfunktio? Miksi? Miksi ei?

Harjoitus 20.8

Määrittele tulostusoperaattori seuraavalle CheckoutRecord-luokalle:

```
class CheckoutRecord {
public:
    // ...
private:
    double book_id;
    string title;
    Date date_borrowed;
    Date date_due;
    pair<string,string> borrower;
    vector< pair<string,string>* > wait_list;
};
```

20.5 Syöttöoperaattorin >> ylikuormitus

Syöttöoperaattorin (>>) ylikuormitus on samanlaista kuin tulostusoperaattorin ylikuormitus paitsi, että virheen mahdollisuus on huomattavasti suurempi. Tässä on esimerkiksi toteutus WordCount-luokan syöttöoperaattorista:

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

/* WordCount-luokkaa pitää muokata niin, että se määrittää syöttöoperaattorin ystäväksi:
  class WordCount {
     friend ostream& operator<<( ostream&, const WordCount& );
     friend istream& operator>>( istream&, WordCount& );
  */

istream&
operator>>( istream &is, WordCount &wd )
{
        /* muoto, jolla WordCount-olio luetaan:
         * <2> string
         *    <7,3> <12,36>
         */

        int ch;

        /* lue sisään pienempi kuin -sananen. ellei löydy,
         * aseta istream virhetilaan ja poistu
         */
        if ((ch = is.get()) != '<' )
        {
                is.setstate( ios_base::failbit );
                return is;
        }

        // lue sisään koon arvo
        int occurs;
        is >> occurs;

        // kaappaa >; ei virhetarkistusta
        while ( is && (ch = is.get()) != '>' ) ;

        is >> wd._word;

        // lue sisään sijaintipaikat;
        // jokaisen sijaintipaikan muoto: < rivi, sarake >
        for ( int ix = 0; ix < occurs; ++ix )
        {
                int line, col;
```

```
                // lue arvot
                while (is && (ch = is.get())!= '<' ) ;
                is >> line;

                while (is && (ch = is.get())!= ',' ) ;
                is >> col;

                while (is && (ch = is.get())!= '>' ) ;

                wd.occurList.push_back( Location( line, col ));
        }
        return is;
}
```

Tässä esimerkissä kuvataan useita asioita, jotka liittyvät mahdolliseen iostream:in virhetilaan:

1. Kun istream epäonnistuu virheellisen muodon takia, sen tulisi merkitä tilansa virheelliseksi (fail):

   ```
   is.setstate( ios_base::failbit )
   ```

2. Kun iostream on virhetilassa, ei lisäys- tai lukuoperaatioilla ole vaikutusta. Esimerkiksi

   ```
   while (( ch = is.get() ) != lbrace)
   ```

   silmukoi ikuisesti, jos istream-olio on virhetilassa. Tästä syystä is:n tila testataan ennen jokaista get()-kutsua:

   ```
   // testaa, onko "is" hyvässä tilassa
   while ( is && (ch = is.get()) != lbrace)
   ```

   Istream-olio, joka ei ole hyvässä tilassa, saa arvokseen epätosi. (Katsomme iostream:in ehtotiloja tarkemmin kohdassa 20.7.)

Seuraava ohjelma lukee aikaisemmin kirjoitetulla WordCount-luokkaoliolla ja tulostaa sen ylikuormitetulla syöttöoperaattorilla, joka määriteltiin edellisessä kohdassa.

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

int main()
{
        WordCount readIn;

        // operator>>( cin, readIn )
        cin >> readIn;

        if ( !cin ) {
                cerr << "WordCount input error" << endl;
                return -1;
        }
```

```
        // operator<<( cout, readIn )
        cout << readIn << endl;
}
```

Kun ohjelma käännetään ja suoritetaan, generoituu seuraava tulostus:

```
<10> rosebud
<11,3>  <11,8>  <14,2>  <34,6>  <49,7>  <67,5>
<81,2>  <82,3>  <91,4>  <97,8>
```

Harjoitus 20.9

WordCount-luokan syöttöoperaattori käsittelee suoraan yksittäisten Location-olioiden syötön. Erota tuo koodi erilliseen Location:in syöttöoperaattoriin.

Harjoitus 20.10

Tee syöttöoperaattori Date-luokalle, joka määriteltiin kohdan 20.4 harjoituksessa 20.7.

Harjoitus 20.11

Tee syöttöoperaattori CheckoutRecord-luokalle, joka määriteltiin kohdan 20.4 harjoituksessa 20.8.

20.6 Tiedoston syöttö ja tulostus

Käyttäjän, joka haluaa yhdistää tiedoston ohjelmaan syöttöä tai tulostusta varten, pitää ottaa mukaan fstream-otsikkotiedosto (joka puolestaan sisältää iostream-otsikkotiedoston):

```
#include <fstream>
```

Avataksemme tiedoston vain lukemista varten, määrittelemme ofstream (tulostustiedostovirran) -luokkaolion. Esimerkiksi:

```
ofstream outfile( "copy.out", ios_base::out );
```

Argumentit, jotka välitetään ofstream:in muodostajalle, määrittävät vuorostaan avattavan tiedoston nimen ja tilan, jossa se avataan. Ofstream-tiedosto voidaan avata joko tulostus- (ios_base::out) tai lisäys- (ios_base::app) -tilassa. (Ostream-tiedosto avataan tulostustilassa oletusarvoisesti.) outfile2:n määrittely on yhtäpitävä outfile:n määrittelyn kanssa:

```
// avataan tulostustilassa oletusarvoisesti
ofstream outfile2( "copy.out" );
```

Jos olemassaoleva tiedosto avataan tulostustilassa, kaikki tallennettu tieto häviää tuosta tiedostosta. Jos sen sijaan haluamme lisätä emmekä korvata olemassa olevaa tiedostoa, tulisi tiedosto avata lisäys tilassa. Kirjoitettu lisätieto sijoittuu tiedoston loppuun. Jos kummassakaan tilassa tiedostoa ei ennestään ole olemassa, se luodaan.

Ennen kuin tiedostoa yritetään lukea tai siihen kirjoittaa, on hyvä varmistua, että sen avaus on onnistunut. Voimme testata outFile-tiedostoa seuraavasti:

```
if ( ! outFile ) { // avaus epäonnistui
    cerr << "cannot open "copy.out" for output\n";
    exit( -1 );
}
```

Ofstream-luokka on johdettu ostream-luokasta. Kaikkia ostream-operaatioita voidaan käyttää ofstream-luokkaolioon. Esimerkiksi

```
char ch = ' ';
outFile.put( '1' ).put( ')' ).put( ch );
outFile << "1 + 1 = " << (1 + 1) << endl;
```

lisää

```
1) 1 + 1 = 2
```

outfile-tiedostoon.

Seuraava ohjelma lukee merkkejä vakiosyötöstä ja kirjoittaa ne tiedostoon copy.out:

```
#include <fstream>

int main()
{
   // avaa tiedosto copy.out tulostusta varten
   ofstream outFile( "copy.out" );

   if ( ! outFile ) {
       cerr << "Cannot open "copy.out" for output\n";
       return -1 ;
   }

   char ch;
   while ( cin.get( ch ) )
                outFile.put( ch );
}
```

Ofstream-luokkajäseneen voidaan käyttää myös käyttäjän määrittelemää tulostusoperaattoria. Seuraava ohjelma käynnistää WordCount:in tulostusoperaattorin, joka määriteltiin edellisessä kohdassa:

```
#include <fstream>
#include "WordCount.h"

int main()
{
   // avaa tiedosto word.out tulostusta varten
   ofstream oFile( "word.out" );

   // onnistuneen avauksen testaus tulee tänne ...
```

```
    // luo ja aseta manuaalisesti
    WordCount artist( "Renoir" );
    artist.found( 7, 12 ); artist.found( 34, 18 );

    // käynnistää operaattorin operator <<(ostream&, const WordCount&);
    oFile << artist;
}
```

Kun tiedosto avataan vain syöttöä varten, käytetään ifstream-luokkaoliota. Ifstream-luokka on johdettu istream-luokasta. Seuraava ohjelma lukee käyttäjän määrittämää tiedostoa ja kirjoittaa sen sisällön vakiotulostukseen:

```
#include <fstream>

int main()
{
    cout << "filename: ";
    string file_name;

    cin >> file_name;

    // avaa tiedosto copy.out syöttöä varten
    ifstream inFile( file_name.c_str() );

    if ( !inFile ) {
        cerr << "unable to open input file: "
             << file_name << " -- bailing out!\n";
        return -1;
    }

    char ch;
    while ( inFile.get( ch ))
          cout.put( ch );
}
```

Seuraava ohjelma lukee Alice Emma -tekstitiedostoa, suodattaa sitä käyttäen filter_string()-funktiota (katso kohdasta 20.2.1 sekä Alice Emma -tekstin että filter_string()-määrittely), lajittelee merkkijonot, poistaa samanlaiset sanat (tuplat), kirjoittaa sitten tuloksena syntyvän tekstin tulostustiedostoon.

```
#include <fstream>
#include <iterator>
#include <vector>
#include <algorithm>

template <class InputIterator>
void filter_string( InputIterator first, InputIterator last,
                    string filt_elems = string("\",?."))
{
```

```
    for ( ; first != last; first++ )
    {
        string::size_type pos = 0;
        while ((pos=(*first).find_first_of(filt_elems,pos))
                != string::npos )
                    (*first).erase( pos, 1 );
    }
}

int main()
{
    ifstream infile( "alice_emma" );

    istream_iterator<string> ifile( infile );
    istream_iterator<string> eos;

    vector< string > text;
    copy( ifile, eos, inserter( text, text.begin() ));

    string filt_elems( "\",.?;:" );
    filter_string( text.begin(), text.end(), filt_elems );

    vector<string>::iterator iter;

    sort( text.begin(), text.end() );
    iter = unique( text.begin(), text.end() );
    text.erase( iter, text.end() );

    ofstream outfile( "alice_emma_sort" );

    iter = text.begin();
    for ( int line_cnt = 1; iter != text.end();
          ++iter, ++line_cnt )
       {
       outfile << *iter << " ";
       if ( ! ( line_cnt % 8 ))
           outfile << '\n';
    }
    outfile << endl;
}
```

Kun ohjelma käännetään ja suoritetaan, generoituu seuraava tulostus:

```
A Alice Daddy Emma Her I Shyly a
alive almost asks at beautiful bird blows but
creature fiery flight flowing hair has he her
him in is it like long looks magical
mean more no red same says she shush
```

```
such tell tells the there through time to
untamed wanting when wind
```

Sekä ifstream- että ofstream-luokkaolio voidaan määritellä määrittämättä tiedostoa. Tiedosto voidaan myöhemmin kytkeä eksplisiittisesti luokkaolioon open()-jäsenfunktion kautta. Esimerkiksi:

```
ifstream curFile;
// ...
curFile.open( filename.c_str() );
if ( ! curFile ) // open failed?
    // ...
```

Tiedosto voidaan kytkeä pois ohjelmasta käynnistämällä close()-jäsenfunktio. Esimerkiksi:

```
curFile.close();
```

Seuraavassa ohjelmassa on avattu ja suljettu viisi tiedostoa vuorollaan käyttäen samaa ifstream-luokkaoliota:

```
#include <fstream>

const int fileCnt = 5;
string fileTabl[ fileCnt ] = {
     "Melville","Joyce","Musil","Proust","Kafka"
};

int main()
{
   ifstream inFile; // ei liitetty mihinkään tiedostoon

   for ( int ix = 0; ix < fileCnt; ++ix )
   {
       inFile.open( fileTabl[ix].c_str() );
       // ... varmista onnistunut avaus
       // ... käsittele tiedosto
       inFile.close();
   }
}
```

Fstream-luokkaolio voi avata tiedoston joko syöttöä *tai* tulostusta varten. Fstream-luokka on johdettu iostream-luokasta. Seuraavassa esimerkissä ensin luetaan ja sitten kirjoitetaan word.out-tiedostoon käyttäen fstream-luokkaoliotiedostoa. Tiedosto word.out, joka luotiin aikaisemmin tässä kohdassa, sisältää WordCount-olion.

```
#include <fstream>
#include "WordCount.h"

int main()
{
   WordCount wd;
   fstream file;
```

```
    file.open( "word.out", ios_base::in );
    file >> wd;
    file.close();

    cout << "Read in: " << wd << endl;

    // ios_base::out hävittäisi nykyisen tiedon
    file.open( "word.out", ios_base::app );
    file << endl << wd << endl;
    file.close();
}
```

Fstream-luokkaolio voi avata tiedoston myös *sekä* syöttöä *että* tulostusta varten. Esimerkiksi seuraava määrittely avaa word.out-tiedoston sekä syöttö- että lisäystilaan:

```
fstream io( "word.out", ios_base::in|ios_base::app );
```

Biteittäistä TAI-operaattoria käytetään määritettäessä useampi kuin yksi tila. Fstream-luokkaolio voidaan positioida uudelleen joko seekg()- tai seekp()-jäsenfunktiolla (g ilmaisee position, josta *luetaan* merkkejä [käytetään ofstream-luokkaolion yhteydessä], kun taas p ilmaisee position, johon *kirjoitetaan* merkkejä [käytetään ifstream-luokkaolion yhteydessä]). Nämä funktiot siirtyvät "absoluuttiseen" osoitteeseen tiedostossa tai tekevät tavusiirtymän tietystä positiosta. Sekä seekg() että seekp() ovat seuraavan muotoisia:

```
// aseta kiinteä positio tiedostoon
seekg( pos_type current_position );

// tietty siirtymä nykyisestä positiosta johonkin suuntaan
seekg( off_type offset_position, ios_base::seekdir dir );
```

Ensimmäisessä versiossa nykyinen positio asetetaan johonkin kiinteään paikkaan tiedostoon, jonka current_position määrittää, jossa 0 on tiedoston alku. Jos tiedosto sisältäisi esimerkiksi merkit

```
abc def ghi jkl
```

seuraava käynnistys

```
io.seekg( 6 );
```

positioi io:n uudelleen merkkipositioon 6; esimerkissämme merkkiin f. Toinen muoto positioi tiedoston uudelleen käyttäen siirtymää joko nykyisestä positiosta tiedoston alkuun tai taaksepäin tiedoston loppuun kuten ilmaistaan toisella argumentilla. dir voidaan asettaa johonkin seuraavista:

1. ios_base::beg, tiedoston alku
2. ios_base::cur, tiedoston nykyinen positio
3. ios_base::end, tiedoston loppu

Seuraavassa esimerkissä jokainen seekg()-käynnistys positioi tiedoston i:nteen tietueeseen

joka iteraatiolla:

```
for ( int i = 0; i < recordCnt; ++i )
    readFile.seekg( i * sizeof(Record), ios_base::beg );
```

Ensimmäiseksi argumentiksi voidaan määrittää myös negatiivinen arvo. Esimerkiksi seuraavassa siirrytään taaksepäin 10 tavua nykyisestä positiosta:

```
readFile.seekg( -10, ios_base::cur );
```

Fstream-tiedoston nykyinen lukupositio voidaan palauttaa jommallakummalla kahdesta jäsenfunktiosta tellg() tai tellp() (jälleen, 'p' ilmaisee *kirjoittamista* — ja palvelee ofstream-oliota; 'g' ilmaisee *lukemista* — ja palvelee ifstream-oliota). Esimerkiksi:

```
// merkitse nykyinen positio
ios_base::pos_type mark = writeFile.tellp();

// ...
if ( cancelEntry )
    // paluu merkittyyn positioon
    writeFile.seekp( mark );
```

Ohjelmoija, joka haluaisi siirtyä yhden Record-tietueen nykyisestä tiedostopositiosta, voisi kirjoittaa jommankumman seuraavista:

```
// yhtäläiset uudelleenpositioinnin käynnistykset
readFile.seekg( readFile.tellg() + sizeof(Record) );

// tätä pidetään tehokkaampana
readFile.seekg( sizeof(Record), ios_base::cur );
```

Käykäämme läpi todellinen ohjelmointiesimerkki hieman tarkemmin. Ongelma on tässä. Meille on annettu tekstitiedosto luettavaksi. Meidän pitää laskea tiedoston tavukoko ja tallentaa se tiedoston loppuun. Lisäksi joka kerta, kun kohtaamme rivinvaihdon, nykyinen tavukoko pitää tallentaa rivinvaihto mukaan lukien tiedoston loppuun. Olkoon esimerkiksi seuraava tekstitiedosto:

```
abcd
efg
hi
j
```

Ohjelman tulisi tuottaa seuraava muokattu tekstitiedosto:

```
abcd
efg
hi
j
5 9 12 14 24
```

Tässä on ensimmäinen toteutuksemme:

```
#include <iostream>
```

```
#include <fstream>

main()
{
   // avaa sekä syöttö- että lisäystilaan
   fstream inOut( "copy.out", ios_base::in|ios_base::app );
   int cnt = 0;   // tavulaskuri
   char ch;

   while ( inOut.get( ch ) )
   {
      cout.put( ch ); // kaiuta päätteelle
      ++cnt;
      if ( ch == '\n' ) {
          inOut << cnt ;
          inOut.put( ' ' ); // tyhjä merkki
      }
   }

   // kirjoita lopullinen tavumäärä
   inOut << cnt << endl;
   cout << "[ " << cnt << " ]" << endl;
   return 0;
}
```

inOut on copy.out-tiedostoon kiinnitetty fstream-luokkaolio, joka on avattu sekä syöttö- että lisäystilaan. Tiedosto, joka avataan lisäystilaan, kirjoittaa kaiken tiedon tiedoston loppuun.

Joka kerta, kun luemme merkin mukaan lukien tyhjän merkin, mutta ei tiedoston loppumerkkiä, kasvatamme cnt-laskuria ja kaiutamme merkin käyttäjän päätteelle. Kaiutuksen tarkoitus on, että meillä on jotakin katsottavaa ja arvioitavaa, jos ohjelmamme ei toimi odotetulla tavalla.

Joka kerta, kun kohtaamme rivinvaihdon, kirjoitamme nykyisen cnt-arvon inOut-tiedostoon. Tiedoston loppumerkin lukeminen päättää silmukan. Kirjoitamme lopullisen cnt-arvon inOut-tiedostoon ja myös näytölle.

Ohjelma kääntyy. Se näyttää toimivan oikein. Tiedosto sisältää muutaman ensimmäisistä lauseista *Moby Dick* -novellista. Se on 1900-luvun amerikkalainen novelli, jonka on kirjoittanut Herman Melville:

> Call me Ishmael. Some years ago, never mind
> how long precisely, having little or no money
> in my purse, and nothing particular to interest
> me on shore, I thought I would sail about a little
> and see the watery part of the world. It is a
> way I have of driving off the spleen, and
> regulating the circulation.

Kun suoritamme ohjelman, generoituu seuraava tulostus:

```
[ 0 ]
```

Yhtään merkkiä ei tulostu ja ohjelma uskoo, että tekstitiedosto on tyhjä. Aivan ilmeistä on, että se on virhe. Olemme ymmärtäneet väärin jotain perustavaa laatua olevan asian. Sen sijaan, että alkaisimme hermoilla tai lannistuisimme liiaksi, pitää nyt miettiä. Ongelma on, että tiedosto on avattu lisäystilaan ja siksi positioidaan sen loppuun. Kun seuraava

```
inOut.get( ch )
```

suoritetaan, kohdataan tiedoston loppumerkki ja while-silmukka päättyy ja jättää cnt-laskurin arvoon 0.

Vaikka ohjelman tulos on vakava virhe, on ratkaisu melko selvä, kun ymmärrämme ongelman. Se, mitä pitää tehdä, on positioida tiedosto takaisin alkuun ennen kuin alamme lukea. Lause

```
inOut.seekg( 0 );
```

tekee juuri sen. Ohjelma käännetään ja suoritetaan uudelleen. Tällä kertaa generoituu seuraava tulostus:

```
Call me Ishmael.  Some years ago, never mind
[ 45 ]
```

Tulostus ja tavumäärä generoituu vain tekstin ensimmäisestä rivistä. Loput kuusi riviä jätetään huomiotta. Hemmetti, kuka on joskus sanonut, että ohjelmointi on helppoa? Tästä huomaa, millaista on olla ohjelmoija silloin, kun on oppimassa jotain uutta. (Joskus on hyödyllistä pitää kirjaa asioista, jotka olemme ymmärtäneet tai tehneet väärin — erityisesti myöhemmin, kun tulemme kärsimättömäksi vähemmän kokeneiden kanssa.) Tässä vaiheessa pitää vetää syvään henkeä ja ajatella, mitä oikein yritämme tehdä ja mitä ilmeisesti olemme tekemässä. Huomaatko ongelman?

Ongelma on, että tiedosto on avattu lisäystilaan. Kun ensimmäisen kerran cnt-tiedostoon kirjoitetaan, tiedosto tulee positioiduksi sen loppuun. Seuraava get() kohtaa tiedoston loppumerkin ja päättää jälleen kerran while-silmukan ennenaikaisesti.

Ratkaisu tällä kertaa on positioida tiedosto uudelleen sinne, missä oltiin ennen cnt:n kirjoittamista. Tämä voidaan toteuttaa seuraavilla kahdella lisälauseella:

```
// merkitse nykyinen positio
ios_base::pos_type mark = inOut.tellg();
inOut << cnt << sp;
inOut.seekg( mark ); // talleta positio
```

Kun ohjelma käännetään ja suoritetaan uudelleen, on tulostus päätteelle oikein. Kun tutkimme tulostustiedostoa, huomaamme, että se ei ole vielä ihan oikein: lopullinen tavumäärää *ei* ole kirjoitettu tiedostoon, vaikka se on kirjoitettu päätteelle. Tulostusoperaattoria, joka tulee while-silmukan jälkeen, ei ole suoritettu.

Ongelma tällä kertaa on, että inOut on “tilassa”, jossa se on kohdannut tiedoston loppumerkin. Niin kauan, kun inOut säilyy tässä tilassa, *ei* syöttö- *eikä* tulostusoperaattoreita suoriteta. Ratkaisu on tyhjentää [clear()] tiedoston tila. Tämä tehdään seuraavalla lauseella:

```
inOut.clear(); // nollaa tilaliput
```

Koko ohjelma näyttää seuraavalta:

```
#include <iostream>
#include <fstream>

int main()
{
   fstream inOut( "copy.out", ios_base::in|ios_base::app );
   int cnt=0;
   char ch;

   inOut.seekg(0);
   while ( inOut.get( ch ) )
   {
      cout.put( ch );
      cnt++;
      if ( ch == '\n' )
      {
         // merkitse nykyinen positio
         ios_base::pos_type mark = inOut.tellg();
         inOut << cnt << ' ';
         inOut.seekg( mark ); // tallenna positio takaisin
      }
   }
   inOut.clear();
   inOut << cnt << endl;

   cout << "[ " << cnt << " ]\n";
   return 0;
}
```

Kun ohjelma käännetään uudelleen ja suoritetaan, generoituu odotettu tulos — lopultakin! Eräs virheistämme ohjelman toteutuksessa oli, että emme tehneet eksplisiittistä lausetta piirteestä, jota sen piti tukea. Jokainen seuraava ratkaisu oli vastaus ongelmaan, joka toi itsensä esille sen sijaan, että olisimme analysoineet koko ongelman, joka vaati ratkaisua. Pääsimme samaan kuin olisimme päässeet miettimällä hieman etukäteen, mutta käytimme siihen huomattavasti enemmän energiaa ja vaivaa.

Harjoitus 20.12

Käyttäen tulostusoperaattoria, joka määriteltiin Date-luokalle harjoituksessa 20.7 tai CheckoutRecord-luokalle harjoituksessa 20.8 (molemmat ovat kohdassa 20.4), kirjoita ohjelma, joka luo ja kirjoittaa tulostustiedostoon.

Harjoitus 20.13

Kirjoita ohjelma, joka avaa ja kirjoittaa tiedostoon, joka luotiin harjoituksessa 20.12. Näytä tiedoston sisältö vakiotulostukseen.

Harjoitus 20.14

Kirjoita ohjelma, joka avaa harjoituksessa 20.12 luodun tiedoston sekä syöttöä että lukemista varten. Tulosta joko Date- tai CheckoutRecord-luokan ilmentymä (a) tiedoston alkuun, (b) toisen olemassaolevan olion jälkeen ja (c) tiedoston loppuun.

20.7 Ehtotilat

Iostream-kirjaston käyttäjinä olemme yleensä huolissamme, onko tiedostovirta virheettömässä tilassa vai ei. Jos kirjoitamme esimerkiksi

```
int ival;
cin >> ival;
```

ja syötämme "Borges", asetetaan cin virhetilaan sen jälkeen, kun on yritetty sijoittaa merkkijonoliteraali kokonaislukuarvoon. Jos kirjoitamme 1024, luku onnistuu ja cin säilyy hyvässä tilassa. Lukuoperaatio suoritetaan vain, jos syöttövirta on hyvässä tilassa.

Jotta voimme päätellä, onko virtaolio hyvässä tilassa, käytämme yleensä yksinkertaista totuusarvoa:

```
if ( !cin )
    // lukuoperaatio epäonnistui tai tiedoston loppumerkki kohdattu
```

Kun luemme tuntemattoman määrän elementtejä, kirjoitamme yleensä seuraavan while-silmukan:

```
while ( cin >> word )
     // ok: lukuoperaatio onnistunut ...
```

while-silmukan ehtotestaus saa arvon epätosi joko saavuttamalla tiedoston loppumerkin tai joutumalla virhetilaan lukuoperaation aikana. Useimmissa tapauksissa tämä stream-olion tosi/epätosi-vertailu on riittävä. Kuitenkin, kun toteutimme Word-syöttöoperaattorin kohdassa 20.5, tarvitsimme hieman hienorakeisempaa pääsyä virran ehtotilaan.

Virtaolio säilyttää joukkoa tilalippuja, joiden kautta virran käyntitilaa voidaan seurata. Seuraavat neljä “väittävää” jäsenfunktiota voidaan käynnistää:

1. eof() palauttaa arvon tosi, jos virta on kohdannut tiedoston loppumerkin. Esimerkiksi:

   ```
   if ( inOut.eof() )
       // ok, kaikki luettu ...
   ```

2. bad() palauttaa arvon tosi, jos tehdään kelpaamaton operaatio kuten positiointiyritys tiedoston lopun jälkeen. Yleensä se ilmaisee, että virta on korruptoitunut jollain tuntemattomalla tavalla.
3. fail() palauttaa arvon tosi, jos operaatio, kuten tiedostovirtaolion avaaminen tai kelpaamattoman syöttömuodon kohtaaminen on epäonnistunut. Esimerkiksi:

   ```
   ifstream iFile( filename, ios_base::in );
   ```

```
if ( iFile.fail() ) // ei pysty avaamaan
  error_message( ... );
```

4. good() palauttaa arvon tosi, jos yksikään muista tiloista ei ole tosi. Esimerkiksi:

```
if ( inOut.good() )
```

On olemassa kaksi metodia, joilla voidaan muokata eksplisiittisesti iostream-olion ehtotilaa. clear()-jäsenfunktiota käyttämällä alustamme ehtotilan pakotettuun arvoon. setstate()-jäsenfunktiolla lisäämme tilan olemassaolevaan olion tilaan sen sijaan, että alustaisimme tilan. Esimerkiksi WordCount-luokkamme syöttöoperaattorissa, kun kohtaamme kelpaamattoman syöttömuodon, käytämme setstate()-jäsenfunktiota ja lisäämme virhetilan istream-olioon:

```
if ((ch = is.get()) != '<' )
{
   is.setstate( ios_base::failbit );
   return is;
}
```

Käytettävissä olevat tila-arvot ovat seuraavat:

```
ios_base::badbit
ios_base::eofbit
ios_base::failbit
ios_base::goodbit
```

Jotta voimme asettaa monitiloja, käytämme biteittäistä TAI-operaattoria kuten seuraavassa:

```
is.setstate( ios_base::badbit | ios_base::failbit );
```

WordCount-syöttöoperaattorin testissä kohdassa 20.5 kirjoitimme

```
if ( !( cin >> readIn ))
{
   cerr << "WordCount input error" << endl;
   exit( -1 );
}
```

Vaihtoehtoisesti olisimme voineet halutessamme jatkaa ohjelmaamme, ehkäpä hälyttäen käyttäjälle syöttövirheestä ja pyytäen lisää syöttötietoa. Jotta kuitenkin voisimme lukea lisää syöttötietoa cin:illä, se pitää palauttaa hyvään tilaan. Voimme tehdä tämän käyttämällä clear()-jäsenfunktiota:

```
cin.clear(); // alustaa cin:in hyvään tilaan
```

Vielä yleisemmin, clear()-jäsenfunktiota käytetään olemassa olevan olion ehtotilan tyhjentämiseen ja nollan tai useamman uuden ehtotilan asettamiseen. Esimerkiksi

```
cin.clear( ios_base::goodbit );
```

palauttaa cin:in eksplisiittisesti hyvään tilaan. (Nämä kaksi käynnistystä ovat yhdenvertaisia, koska clear()-käynnistyksen oletusarvo on goodbit-arvo.)

rdstate()-jäsenfunktio mahdollistaa, että voimme käsitellä eksplisiittisesti iostream-luokka-olion tilaa. Esimerkiksi:

```
ios_base::iostate old_state = cin.rdstate();

cin.clear();
process_input();

// aseta cin nyt uudelleen vanhaan tilaan
cin.clear( old_state );
```

Harjoitus 20.15

Uudista joko (tai molempien) Date-luokan harjoituksen 20.7 ja/tai CheckoutRecord-luokan harjoituksen 20.8 (molemmat kohdassa 20.4) syöttöoperaattori niin, että se asettaa istream-olion ehtotilan. Muokkaa ohjelmia, joita käytettiin operaattorin kokeiluun niin, että ne tarkistavat eksplisiittisesti asetetut tilat ja kun ne ovat raportoineet, alustavat istream-olion ehtotilan. Kokeile uudistettua ohjelmaa antamalla sekä hyviä että huonoja syöttömuotoja.

20.8 String-virrat

Iostream-kirjasto tukee muistissa tapahtuvia operaatioita string-olioilla. Ostringstream-luokka lisää merkkejä string-olioon. Istringstream-luokka lukee merkkejä string-oliosta ja string-stream-luokkaa voidaan käyttää sekä lukemisen että kirjoittamisen tukena. Jotta voimme käyttää string-virtaa, pitää ottaa mukaan siihen liittyvä otsikkotiedosto:

```
#include <sstream>
```

Esimerkiksi seuraava funktio lukee koko alice_emma-tiedoston ostringstream-luokkaolioon buf. Puskuri buf kasvaa tarvittaessa merkkisyötön mukaisesti:

```
#include <string>
#include <fstream>
#include <sstream>

string read_file_into_string()
{
        ifstream ifile( "alice_emma" );
        ostringstream buf;

        char ch;
        while ( buf && ifile.get( ch ))
              buf.put( ch );

        return buf.str();
}
```

str()-jäsenfunktio palauttaa string-olion, joka liittyy ostringstream-luokkaolioon. Tätä string-oliota voidaan nyt käsitellä samalla tavalla kuin mitä tahansa “tavallista” string-oliota.

Esimerkiksi seuraavassa ohjelmassa text alustetaan jäsenittäin string-oliolla, joka liittyy buf-puskuriin:

```
int main()
{
        string text = read_file_into_string();

        // merkitsee jokaisen rivinvaihdon position tekstissä
        vector< string::size_type > lines_of_text;
        string::size_type pos = 0;

        while ( pos != string::npos )
        {
                pos = text.find( '\n', pos );
                lines_of_text.push_back( pos );
        }

        // ...
}
```

Ostringstream-oliota voidaan käyttää myös yhdistetyn string-olion automaattiseen muotoiluun; tarkoittaa, että string-olio muodostuu useista tietotyypeistä. Esimerkiksi tulostusoperaattori konvertoi automaattisesti minkä tahansa aritmeettisen tyypin vastaavaksi merkkijonoesitystavaksi ilman, että meidän pitää huolehtia tarvittavasta muistimäärästä:

```
#include <iostream>
#include <sstream>

int main()
{
    int ival    = 1024;    int *pival    = &ival;
    double dval = 3.14159; double *pdval = &dval;

    ostringstream format_message;

    // ok: konvertoi arvoja merkkijonoesitystapaan
    format_message << "ival: "  << ival
        << " ival's address: " << pival << ô\nô
        << "dval: " << dval
        << " dval's address: " << pdval << endl;

    string msg = format_message.str();
    cout << " size of message string: " << msg.size()
        << " message: " << msg << endl;

}
```

Joissakin tilanteissa on parempi kerätä vähempiarvoisia diagnostisia virheitä ja varoituksia sen sijaan, että näyttäisi ne heti kohdattaessa. Yksinkertainen tapa toteuttaa tämä on tehdä yli-

kuormitettu joukko muotoilufunktioita yleisessä muodossa

```
string
format( string msg, int expected, int received )
{
    ostringstream message;
    message << msg << " expected: " << expected
          << " received: " << received << "\n";
    return message.str();
}

string format( string msg, vector<int> *values );
// ... jne.
```

Sovellus voi sitten tallentaa merkkijonot myöhempää tulostusta varten, ehkäpä luokitellen ne vakavuuden mukaan. Tämä voidaan yleistää Notify-, Log- tai Error-luokkaan.

Istringstream lukee sille muodostettua string-oliota. Eräs istringstream:in käyttötapa on konvertoida numeerinen merkkijonoesitystapa aritmeettiseksi arvoksi. Esimerkiksi:

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

int main()
{
        int ival  = 1024;  int *pival  = &ival;
        double dval = 3.14159; double *pdval = &dval;

        // luo merkkijonon, jossa jokainen arvo
        // on tallennettu tyhjällä merkillä erotettuna toisistaan
        ostringstream format_string;
        format_string << ival << " " << pival << " "
                << dval << " " << pdval << endl;

        // lukee tallennetut ascii-arvot sijoittaen
        // ne vuorollaan neljään olioon
        istringstream input_istring( format_string.str() );
        input_istring >> ival >> pival
                >> dval >> pdval;
}
```

Harjoitus 20.16

C:ssä tulostettava ilmoitus muotoillaan standardilla C printf() -perheen rutiineilla. Esimerkiksi seuraava koodikatkelma

```
int    ival = 1024;
double dval = 3.14159;
char   cval = 'a';
char  *sval = "the end";
```

```
printf( "ival: %d\tdval: %g\tcval: %c\tsval: %s",
        ival, dval, cval, sval );
```

generoi

```
ival: 1024dval: 3.14159cval: asval: the end
```

printf():n ensimmäinen argumentti on muotoilumerkkijono. Jokainen %-merkki ilmaisee, että tuo paikka korvataan argumentin arvolla; sen jälkeen tuleva merkki ilmaisee sen tyypin. Tässä on joitakin mahdollisia tuettuja tyyppejä:

%d	integer
%g	floating point
%c	char
%s	C-style string

(Katso KERNIGHAN88] nähdäksesi aiheen koko käsittelyn.)

printf():n lisäargumentit täsmätään paikkasidonnaisesti jokaiseen %-muotoiluparin ilmentymään. Muut muotoilumerkkijonon merkit tulkitaan literaaleina ja kirjoitetaan sellaisenaan suoraan.

printf()-rutiiniperheen kaksi pääheikkoutta ovat, että (1) muotoilumerkkijono ei havaitse käyttäjän määrittelemiä tyyppejä ja (2) jos argumenttien tyyppi tai lukumäärä ei täsmää muotoilumerkkijonoon, virhettä ei havaita, ja tulostus on huonosti muotoiltu. printf()-rutiiniperheen päävetovoima on muotoilumerkkijonostrategian tiiviys.

(a) Generoi samanlainen muotoiltu tulostus ostringstream-oliota käyttäen.

(b) Aseta vastakkain näiden kahden eri lähestymistavan etuja ja huonoja puolia.

20.9 Muotoilutila

Jokainen iostream-kirjaston luokkaolio pitää yllä *muotoilutilaa* (*format state*), joka ohjaa muotoiluoperaatioiden yksityiskohtia kuten kokonaisarvon kantaluvun merkintätapaa ja liukulukuarvon tarkkuutta. Ohjelmoijalla on käytettävissään esimääriteltyjä manipulaattoreita olion[1] muotoilutilan muokkaamiseen.

1. Lisäksi ohjelmoija voi asettaa tai ottaa asetuksen pois muotoilutilan lipuista suoraan käyttäen setf()- ja unsetf()-jäsenfunktioita. Emme käsittele näitä operaatioita. Nähdäksesi tämän käsittelyn tarkemmin, katso julkaisusta [STROUSTRUP97].

Manipulaattoria käytetään virtaolioon aivan kuin se olisi tietoa. Sen sijaan, että se saisi aikaan tiedon luvun tai kirjoittamisen, manipulaattori muokkaa virtaolion sisäistä tilaa. Esimerkiksi oletusarvo on, että bool-olion arvo true (ja literaalivakio true) tulostetaan kokonaislukuarvona 1:

```
#include <iostream>

int main()
{
   bool illustrate = true;;
   cout << "bool object illustrate set to true: "
        << illustrate << '\n';
}
```

Jotta voisimme muokata cout:ia niin, että illustrate tulostetaan sanana true, käytämme boolalpha-manipulaattoria:

```
#include <iostream>

int main()
{
   bool illustrate = true;;
   cout << "bool object illustrate set to true: ";

   // muuta cout:in tilaa niin, että se tulostaa boolean-arvot
   // käyttäen merkkijonoja true ja false.
   cout << boolalpha;
   cout << illustrate << '\n';
}
```

Koska manipulaattorin käyttö palauttaa virtaolion, johon sitä on käytetty, voimme yhdistää sen palautusarvon tiedon kanssa (ja muita manipulaattoreita myös). Tässä on pieni ohjelmamme kirjoitettu uudelleen, ja se sekoittaa keskenään tietoa ja manipulaattoreita:

```
#include <iostream>

int main()
{
   bool illustrate = true;
   cout << "bool object illustrate: "
       << illustrate
       << "\nwith boolalpha applied: "
       << boolalpha << illustrate << '\n';

   // ...
}
```

Manipulaattoreiden sekoittaminen tiedon kanssa kuten tässä on hieman harhaanjohtavaa. Manipulaattorin käyttö ei vain muuta seuraavan tulostuksen esitystapaa. Sen sijaan ostream-olion sisäinen muotoilutila muuttuu. Esimerkissämme kaikki bool-arvot ohjelman loppuun saakka näytetään myös joko true- tai false-arvoina.

Ottaaksemme cout:in muokatun asetuksen pois käytöstä, pitää käyttää noboolalpha-manipulaattoria:

```
cout << boolalpha // asettaa cout:in sisäisen tilan
   << illustrate
   << noboolalpha // ottaa cout:in sisäisen asetuksen pois päältä
```

Kuten näemme, monet manipulaattoreista tulevat aseta/palauta-pareina.

Oletusarvo on, että aritmeettiset arvot kirjoitetaan ja luetaan desimaalisella merkintätavalla. Ohjelmoija saattaa muuttaa kokonaisarvojen kantaluvun merkintätapaa oktaaliseksi tai heksadesimaaliseksi tai takaisin desimaaliseksi (liukulukuarvojen esitystapa säilyy muuttumattomana) käyttämällä manipulaattoreita hex, oct ja dec. Esimerkiksi:

```
#include <iostream>
int main()
{
        int ival = 16;
        double dval = 16.0;

        cout << "ival: " << ival
            << " oct set: " << oct << ival << "\n";

        cout << "dval: " << dval
            << " hex set: " << hex << dval << "\n";

        cout << "ival: " << ival
            << " dec set: " << dec << ival << "\n";
}
```

Kun ohjelma käännetään ja suoritetaan, generoituu seuraava tulostus:

```
ival: 16 oct set: 20
dval: 16 hex set: 16
ival: 10 dec set: 16
```

Ohjelmamme eräs ongelma on, että emme voi katsoa arvoa ja olla varmoja sen kantaluvun merkintätavasta. Onko esimerkiksi 20 todella 20 vai oktaalinen esitystapa arvosta 16? Manipulaattori showbase saa aikaan kokonaisarvon tulostukseen sen kantaluvun seuraavin käytännöin:

1. Edessä oleva 0x ilmaisee heksadesimaalia (jos haluamme X:n tulostuvan isona kirjaimena kuten X, voimme käyttää uppercase-manipulaattoria; paluu takaisin pieneen x-kirjaimeen tapahtuu käyttämällä nouppercase-manipulaattoria).

2. Edessä oleva 0 ilmaisee oktaalia.
3. Kummankin edellisen puuttuminen ilmaisee desimaalia.

Tässä on uudistettu ohjelmamme, jossa on käytetty showbase-manipulaattoria:

```
#include <iostream>
int main()
{
        int ival = 16;
        double dval = 16.0;

        cout << showbase;

        cout << "ival: " << ival
            << " oct set: " << oct << ival << "\n";

        cout << "dval: " << dval
            << " hex set: " << hex << dval << "\n";

        cout << "ival: " << ival  << " dec set: "
            << dec << ival << "\n";

        cout << noshowbase;
}
```

Tässä on uudistettu tulostus:

```
ival: 16 oct set: 020
dval: 16 hex set: 16
ival: 0x10 dec set: 16
```

noshowbase-manipulaattori alustaa cout:in alkuarvoonsa, jotta se ei näytä kokonaisarvojen kantaluvun merkintää.

Oletusarvo on, että liukulukuarvon tarkkuus on 6. Tätä arvoa voidaan muokata käyttämällä joko precision(int)-jäsenfunktiota tai setprecision()-virtamanipulaattoria (kun jälkimmäistä käytetään, pitää ottaa mukaan iomanip-otsikkotiedosto). precision() palauttaa nykyisen tarkkuusarvon. Esimerkiksi:

```
#include <iostream>
#include <iomanip>
#include <math.h>

int main()
{
   cout  << "Precision: "
        << cout.precision() << endl
        << sqrt(2.0) << endl;

        cout.precision(12);
        cout  << "\nPrecision: "
```

```
                << cout.precision() << endl
                << sqrt(2.0) << endl;

        cout  << "\nPrecision: " << setprecision(3)
                << cout.precision() << endl
                << sqrt(2.0) << endl;

        return 0;
}
```

Kun ohjelma käännetään ja suoritetaan, generoituu seuraava tulostus:

```
Precision: 6
1.41421

Precision: 12
1.41421356237

Precision: 3
1.41
```

Manipulaattorit, jotka saavat argumentin kuten setprecision() ja setw(), jota katsoimme aikaisemmin, vaativat, että otamme mukaan iomanip-otsikkotiedoston:

```
#include <iomanip>
```

Esimerkkimme ei näytä kahta lisänäkökohtaa setprecision()-funktiosta: (1) kokonaisarvot säilyvät muuttumattomina ja (2) liukulukuarvot pyöristetään, mutta ei katkaista. Täten luvusta 3.14159 tulee 3.142, kun tarkkuus on asetettu arvoon 4, ja 3.14, kun tarkkuus on asetettu arvoon 3.

Oletusarvo on, että desimaalipistettä ei näytetä, kun murto-osan arvo on 0. Esimerkiksi

```
cout << 10.00
```

tulostaa

```
10
```

Pakottaaksemme desimaalipisteen näkyviin, käytämme showpoint-manipulaattoria:

```
cout << showpoint
   << 10.0
   << noshowpoint << '\n';
```

noshowpoint-manipulaattori asettaa jälleen oletuskäyttäytymisen.

Jälleen oletusarvo on, että liukulukuarvot näytetään kiinteällä desimaali-ilmaisulla. Vaihtaaksemme tieteellisen ilmaisun, käytämme scientific-manipulaattoria. Vaihtaaksemme sen takaisin kiinteäksi desimaaliseksi, käytämme fixed-manipulaattoria:

```
cout << "scientific: " << scientific
   << 10.0
   << "fixed decimal: " << fixed
   << 10.0 << '\n';
```

Tämä generoi

```
scientific: 1.0e+01
fixed decimal: 10
```

Jos haluamme, että ôe' näkyy tieteellisellä ilmaisulla ôE', käytämme uppercase-manipulaattoria. Palataksemme takaisin pieneen kirjaimeen, käytämme nouppercase-manipulaattoria. (uppercase-manipulaattori ei saa aikaan kaikkien aakkosmerkkien näkymistä isona kirjaimena!)

Oletusarvo on, että ylikuormitetut syöttöoperaattorit jättävät lukematta tyhjän merkin (välilyönti, rivinvaihto, sivunvaihto ja vaununpalautus). Olkoon merkkijono

```
a b  c
d
```

Silmukka

```
char ch;
while ( cin >> ch )
      // ...
```

toistaa neljä kertaa ja lukee merkit väliltä a - d jättäen lukematta väliin jäävät tyhjät merkit, mahdolliset sarkain- ja rivinvaihtomerkit. Manipulaattori noskipws saa aikaan sen, että syöttöoperaattori ei jätä lukematta tyhjää merkkiä:

```
char ch;
cin >> noskipws;
while ( cin >> ch )
      // ...
cin >> skipws;
```

while-silmukka tarvitsee nyt seitsemän toistokertaa, että se voi lukea merkit väliltä a - d. Jos halutaan palata oletuskäyttäytymiseen, käytetään skipws-manipulaattoria cin:iin.

Kun kirjoitamme

```
cout << "please enter a value: ";
```

literaalimerkkijono tallennetaan cout:in puskuriin. On lukuisia tilanteita, jotka saavat aikaan esimerkissämme puskurin “huuhtelun” — eli tyhjentämisen — kun kirjoitetaan vakiosyöttöön:

1. Puskuri voi tulla täyteen. Siinä tapauksessa se pitää huuhdella, jotta uusi arvo voidaan lukea.
2. Voimme huuhdella puskurin eksplisiittisesti käyttämällä joko flush-, ends- tai endl-manipulaattoria:

```
// huuhtelee puskurin
cout << "hi!" << flush;
```

```
// lisää nollan, sitten huuhtelee puskurin
char ch[2]; ch[0] = 'a'; ch[1] = 'b';
cout << ch << ends;

// lisää rivinvaihdon, sitten huuhtelee puskurin
cout << "hi!" << endl;
```

3. unitbuf, joka on sisäinen virran tilamuuttuja, voidaan asettaa tyhjentämään puskuri jokaisen tulostusoperaation jälkeen.
4. Ostream-olio voidaan *sitoa* istream-olioon, jolloin ostream-puskuri huuhdellaan aina, kun istream lukee syöttövirtaa. cout on esimääritellysti sidottu cin:iin:

   ```
   cin.tie( &cout );
   ```

   Lause

   ```
   cin >> ival;
   ```

saa aikaan cout:iin liittyvän puskurin huuhtelun.

Ostream-olio voidaan sitoa vain yhteen istream-olioon kerralla. Katkaistaksemme olemassaolevan sidoksen, välitämme 0-argumentin. Esimerkiksi:

```
istream is;
ostream new_os;

// ...

// tie() palauttaa olemassaolevan sidoksen
ostream *old_tie = is.tie();

is.tie( 0 ); // katkaisee olemassaolevan sidoksen
is.tie( &new_os ); // aseta uusi sidos

// ...

is.tie( 0 ); // katkaise olemassaoleva sidos
is.tie( old_tie ); // palauta vanha sidos
```

Voimme ohjata numeerisen tai merkkijonon arvon leveyttä tulostuksessa setw()-manipulaattorilla. Esimerkiksi ohjelma

```
#include <iostream>
#include <iomanip>

int main()
{
    int ival = 16;
    double dval = 3.14159;

    cout << "ival: " << setw(12) << ival << '\n'
```

```
                << "dval: " << setw(12) << dval << '\n';
}
```

generoi seuraavan tulostuksen:

```
ival:        16
dval:     3.14159
```

Toinen setw() on välttämätön, koska toisin kuin muut manipulaattorit, setw() ei muokkaa ostream-olion muotoilutilaa.

Jotta saisimme arvon asetettua vasempaan laitaan, käytämme left-manipulaattoria (right-manipulaattori asettaa sen uudelleen oletusarvoon). Jos haluamme generoida

```
  16
-  3
```

käytämme internal-manipulaattoria, joka asettelee etumerkin vasemmalle ja arvon oikealle sijoittaen niiden väliin tyhjiä merkkejä. Jos haluamme täyttää tyhjät merkit jollakin toisella merkillä, käytämme setfill()-manipulaattoria.

```
cout << setw(6) << setfill('%') << 100 << endl;
```

generoi

```
%%%100
```

Kaikki esimääritellyt manipulaattorit on lueteltu taulukossa 20.1.

Taulukko 20.1 Manipulaattorit

Manipulaattori	Tarkoitus
boolalpha	esitä tosi (true) ja epätosi (false) merkkijonoina
*noboolalpha	esitä tosi ja epätosi arvoina 0 ja 1
showbase	generoi etuliite, joka ilmaisee numeerisen kantaluvun
*noshowbase	älä generoi kantaluvun etuliitettä
showpoint	näytä aina desimaalipiste
*noshowpoint	näytä desimaalipiste vain murto-osan yhteydessä
showpos	näytä + ei-negatiivisten numeroiden yhteydessä
*noshowpos	älä näytä + ei-negatiivisten numeroiden yhteydessä

Jatkuu

Taulukko 20.1 Manipulators (jatkuu)

Manipulaattori	Tarkoitus
*skipws	jätä tyhjät merkit lukematta syöttöoperaattoreilla
noskipws	älä jätä tyhjiä merkkejä lukematta syöttöoperaattoreilla
uppercase	tulosta 0X heksadesimaalisessa, E tieteellisessä
*nouppercase	tulosta 0x heksadesimaalisessa, e tieteellisessä
*dec	näytä kymmenjärjestelmänä
hex	näytä heksadesimaalisena
oct	näytä oktaalisena
left	lisää täytemerkit arvosta oikealle
right	lisää täytemerkit arvosta vasemmalle
internal	lisää täytemerkit etumerkin ja arvon välille
*fixed	näytä liukuluku kymmenjärjestelmänä
scientific	näytä liukuluku tieteellisenä
flush	huuhtele ostream-puskuri
ends	lisää null-merkki, sitten huuhtele ostream-puskuri
endl	lisää rivinvaihtomerkki, sitten huuhtele ostream-puskuri
ws	“syö” tyhjä merkki
// nämä vaativat #include <iomanip>	
setfill(ch)	täytä tyhjä merkki merkillä ch
setprecision(n)	aseta liukulukutarkkuus arvoon n
setw(w)	lue tai kirjoita arvo w merkkiin
setbase(b)	tulosta kokonaisluvut kantaluvulla b
* ilmaisee virtatilan oletusarvoa	

20.10 Vahvasti tyypitetty kirjasto

Iostream-kirjasto on vahvasti tyypitetty. Kun esimerkiksi yritetään lukea ostream:ia tai sijoittaa ostream istream:iin, saadaan molemmat virhetilanteet siepattua käännöksen aikana tyyppirikkomusvirheinä. Olkoon esimerkiksi seuraavat esittelyt:

```
#include <iostream>
#include <fstream>
class Screen;

extern istream& operator>>( istream&, const Screen& );
extern void print( ostream& );
ifstream inFile;
```

Seuraavat kaksi lausetta johtavat käännöksen aikana tyyppivirheeseen:

```
int main()
{
   Screen myScreen;

   // virhe: edellyttää ostream&:ia
   print( cin >> myScreen );

   // virhe: edellyttää >>-operaattoria
   inFile << "error: output operator";
}
```

Syöttö- ja tulostuspiirteet ovat komponenttina C++-vakiokirjastossa. Luvussa 20 ei kuvata koko iostream-kirjastoa — erityisesti käyttäjän määrittelemien manipulaattorien ja puskuriluokkien luonti ovat tämän kirjan aihepiirin ulkopuolella. Sen sijaan olemme keskittyneet siihen osaan iostream-kirjastoa, joka on ohjelman syötön ja tulostuksen kannalta olennaista.