Osa V

Oliokeskeinen ohjelmointi

Oliokeskeinen ohjelmointi (*object-oriented programming*) laajentaa oliopohjaista ohjelmointia tarjoten tyyppi/alityyppi-suhteita. Tämä saavutetaan mekanismin kautta, jota kutsutaan *periytymiseksi* (*inheritance*). Sen sijaan, että toteutettaisiin uudelleen yhteisiä piirteitä, luokka perii tietojäsenet ja jäsenfunktiot yliluokaltaan (*parent class*). C++:n periytymistä tukee mekanismi, jota sanotaan *luokasta johtamiseksi* (*class derivation*). Luokkaa, josta periydytään, sanotaan *kantaluokaksi* (*base class*). Uutta luokkaa sanotaan *johdetuksi luokaksi* (*derived class*). Kutsumme kantaluokan ja johdettujen luokkien instantiointeja luokkaperiytymisen *hierarkiaksi*.

Esimerkiksi 3D-tietokonegrafiikassa sekä OrthographicCamera että PerspectiveCamera on yleensä johdettu abstraktista Camera-kantaluokasta. Operaatiot ja tiedot, jotka ovat yhteisiä kaikille kameroille, on määritelty abstraktiin Camera-luokkaan. Jokainen johdettu luokka toteuttaa vain omat eronsa abstraktiin Camera-luokkaan joko tekemällä vaihtoehtoiset toteutukset perityistä jäsenfunktioista tai esittelemällä lisäjäseniä.

Jos kantaluokka ja johdetut luokat jakavat saman julkisen rajapinnan, sanotaan johdetun luokan olevan kantaluokkansa *alityyppi*. Esimerkiksi PerspectiveCamera on Camera-luokan alityyppi. C++:ssa vallitsee erityinen tyyppi/alityyppi-suhde, jossa kantaluokan osoitin tai viittaus voi osoittaa mitä tahansa siitä johdettuja luokan alityyppejä ilman ohjelmoijan väliintuloa. (Tätä kykyä käsitellä useampaa kuin yhtä tyyppiä osoittimella tai viittauksella kantaluokkaan sanotaan *monimuotoisuudeksi* (*polymorphism*)). Olkoon esimerkiksi funktio

```
void lookAt( const Camera *pcamera );
```

niin toteutamme `lookAt()`-funktion ohjelmoimalla Camera-kantaluokan rajapinnan riippumattomaksi siitä, viittaako `pcamera` PerspectiveCamera-, OrthographicCamera- tai johonkin tulevaisuudessa johdettavaan Camera-alityyppiin, joka on vielä tuntematon.

Jokainen yksittäinen `lookAt()`-käynnistys saa yhden Camera-alityyppiolion osoitteen. Kääntäjä konvertoi sen automaattisesti sopivaksi kantaluokan osoittimeksi. Esimerkiksi:

```
// ok: konvertoitu automaattisesti tyypiksi Camera*
OrthographicCamera ocam;
lookAt( &ocam );

// ...

// ok: konvertoitu automaattisesti tyypiksi Camera*
PerspectiveCamera *pcam = new PerspectiveCamera;
lookAt( pcam );
```

lookAt()-toteutuksemme on suojattu sovelluksemme todellisilta Camera-alityypeiltä. Jos myöhemmin haluaisimme lisätä tai poistaa alityypin, ei lookAt()-toteutustamme tarvitsisi muuttaa.

Alityyppien monimuotoisuus mahdollistaa, että voimme kirjoittaa sovelluksemme ydinalueen riippumattomaksi yksittäisistä tyypeistä, joita haluamme käsitellä. Sen sijaan ohjelmoimme abstraktiomme kantaluokan julkisen rajapinnan kantaluokan osoittimien ja viittausten kautta. Todellisuudessa välitetty tyyppi ratkaistaan suorituksen aikana ja vastaava julkisen rajapinnan ilmentymä käynnistetään.

Käynnistettävän funktion suorituksenaikaista ratkaisua sanotaan *dynaamiseksi sitomiseksi* (*dynamic binding*) (oletusarvo on, että funktiot ratkaistaan *staattisesti* käännöksen aikana). C++:ssa dynaamista sitomista tuetaan mekanismin kautta, jota sanotaan *virtuaalifunktioksi*. Alityyppien monimuotoisuus periytymisen ja dynaamisen sitomisen kautta muodostavat oliokeskeisen ohjelmoinnin perustan, joka on seuraavien lukujen aiheena.

Luku 17 kattaa C++:n piirteet, jotka tukevat oliokeskeistä ohjelmointia ja tutkii, kuinka periytyminen vaikuttaa luokkamekanismeihin kuten muodostaja, tuhoaja ja jäsenittäinen alustaminen ja sijoittaminen. Jotta käsittelyä voitaisiin elävöittää, Query-luokkahierarkiaa kehitetään luvussa 6 esitellyn tekstinkyselyjärjestelmän tuella.

Luvussa 18 tutkitaan monimutkaisempia periytymishierarkioita, jotka ovat mahdollisia moni- ja virtuaaliperiytymisen kautta. Luvussa laajennetaan luvun 16 luokkamallia kolmetasoiseksi mallihierarkiaksi käyttämällä moni- ja virtuaaliperiytymistä.

Luvussa 19 käsitellään suorituksenaikaista tyyppitunnistusta (Run-Time Type Identification, RTTI), ja siinä on myös syvältä luotaava katsaus ylikuormitetun funktion ratkaisun tuesta periytymisen kannalta. Luvussa myös palataan uudelleen poikkeusten käsittelypiirteisiin ja käsitellään vakiokirjaston poikkeusluokkahierarkiaa ja kuvataan, kuinka omia poikkeusluokkia määritellään ja käsitellään.

Luvussa 20 nähdään syvältä luotaava käsittely iostream-kirjastosta. Iostream-kirjasto on luokkahierarkia, joka tukee sekä virtuaali- että moniperiytymistä.

17

Luokkaperiytyminen ja alityypitys

Motivoidaksemme ja kuvataksemme abstraktien säiliötyyppien käsittelyä luvussa 6, kävimme läpi tekstinkyselyjärjestelmän osittaisen toteutuksen, jonka kapseloimme viimeisessä vaiheessa TextQuery-luokaksi. Jätimme sen kuitenkin viimeistelyä vaille ja lykkäsimme varsinaisen käyttäjäkyselyn siihen saakka, kunnes olisimme käsitelleet oliokeskeisen ohjelmoinnin. Tässä luvussa toteutamme käyttäjän kyselykielen yksiperiytymisen kautta Query-luokkahierarkiana tuodaksemme esille lyhyen katsauksen oliokeskeisestä suunnittelusta ja ohjelmoinnista C++:ssa. Lisäksi muokkaamme ja laajennamme luvun 6 TextQuery-luokkaamme, jotta saisimme siitä täysin integroidun tekstinkyselyjärjestelmän.

Ohjelma, jolla tekstinkyselyjärjestelmäämme ajetaan, on seuraavassa:

```
#include "TextQuery.h"

int main()
{
    TextQuery tq;

    tq.build_text_map();
    tq.query_text();
}
```

build_text_map() on hieman muunneltu muoto luvun 6 doit()-jäsenfunktiosta. Sen päätehtävä on rakentaa sanojen sijaintikartta, jota indeksoidaan tekstissä olevalla jokaisella merkitsevällä sanalla. (Jos muistat, emme tallenna merkitykseltään neutraaleja sanoja kuten if, and, but jne. Lisäksi poistamme isot kirjaimet ja käsittelemme monikkomuotojen loppuliitteet kuten esimerkiksi testifies muotoon testify ja marches muotoon march.) Jokaiseen sanaan liittyy paikkavektori, jossa vektorin jokaiseen elementtiin tallennetaan tekstin sanan ilmentymän rivi ja sarake.

query_text() pyytää ja muuntaa käyttäjän jokaisen kyselyn sisäiseksi oliokeskeiseksi Query-luokkahierarkiaksi käyttäen hyväkseen yksiperiytymistä ja dynaamista sitomista. Sisäistä kyselyn esitystapaa arvioidaan sanojen sijaintikarttaa vasten, jonka on luonut build_text_map(). Ratkaisu on yksilöllinen rivijoukko tekstitiedostossa, joka tyydyttää kyselyn kriteerit. Esimerkiksi:

```
Enter a query-please separate each item by a space.
Terminate query (or session) with a dot( . ).

==> fiery && ( bird || shyly )

     fiery ( 1 ) lines match
     bird ( 1 ) lines match
     shyly ( 1 ) lines match
      ( bird || shyly )  ( 2 ) lines match
     fiery &&  ( bird || shyly )  ( 1 ) lines match

Requested query: fiery &&  ( bird || shyly )

( 3 ) like a fiery bird in flight. A beautiful fiery bird, he tells her,
```

Kyselypiirre, jota tuemme, muodostuu seuraavista elementeistä:

1. Yksittäinen sana kuten Alice tai untamed. Kaikki rivit, joissa sana esiintyy, näytetään suluissa olevan rivinumeron kanssa. (Rivit näytetään nousevassa järjestyksessä.) Esimerkiksi:

```
==> daddy
.
   daddy ( 3 ) lines match

Requested query: daddy

( 1 ) Alice Emma has long flowing red hair. Her Daddy says
( 4 ) magical but untamed. "Daddy, shush, there is no such thing,"
( 6 ) Shyly, she asks, "I mean, Daddy, is there?"
```

2. Ei-kysely, käytetään !-operaattoria. Kaikki rivit, joissa nimi ei esiinny, näytetään. Esimerkiksi tässä jätetään 1-kohdan sana pois:

```
==> ! daddy
.
     daddy ( 3 ) lines match
      ! daddy ( 3 ) lines match

Requested query:  ! daddy

( 2 ) when the wind blows through her hair, it looks almost alive,
( 3 ) like a fiery bird in flight. A beautiful fiery bird, he tells her,
( 5 ) she tells him, at the same time wanting him to tell her more.
```

3. Tai-kysely, käytetään ||-operaattoria. Kaikki rivit, joissa esiintyy jompikumpi kahdesta nimestä, näytetään. Esimerkiksi:

```
==> fiery || untamed
.
     fiery ( 1 ) lines match
```

```
    untamed ( 1 ) lines match
    fiery || untamed ( 2 ) lines match

Requested query: fiery || untamed

( 3 ) like a fiery bird in flight. A beautiful fiery bird, he tells her,
( 4 ) magical but untamed. "Daddy, shush, there is no such thing,"
```

4. Ja-kysely, käytetään &&-operaattoria. Kaikki rivit, joissa molemmat sanat ovat ja lisäksi vierekkäin, näytetään. Tähän kuuluvat rivin viimeinen sana ja seuraavan rivin ensimmäinen sana. Esimerkiksi:

```
==> untamed && Daddy
.
    untamed ( 1 ) lines match
    daddy ( 3 ) lines match
    untamed && daddy ( 1 ) lines match

Requested query: untamed && daddy

( 4 ) magical but untamed. "Daddy, shush, there is no such thing,"
```

Nämä elementit voidaan yhdistää kuten tässä

```
fiery && bird || shyly
```

Kuitenkin arviointijärjestys on vasemmalta oikealle, siten että jokainen elementti säilyttää saman sidontatason. Joten edellisen yhdistetyn kyselyn arviointi vastaa osumia `fiery bird` tai `shyly` eikä `fiery bird` tai `fiery shyly`:

```
==> fiery && bird || shyly
.
    fiery ( 1 ) lines match
    bird ( 1 ) lines match
    fiery && bird ( 1 ) lines match
    shyly ( 1 ) lines match
    fiery && bird || shyly ( 2 ) lines match

Requested query: fiery && bird || shyly

( 3 ) like a fiery bird in flight. A beautiful fiery bird, he tells her,
( 6 ) Shyly, she asks, "I mean, Daddy, is there?"
```

Jotta voisimme mahdollistaa kyselyn aliryhmityksen, pitää kyselymme tukea myös sulkuja. Esimerkiksi:

```
fiery && ( bird || shyly )
```

etsii kaikki viittaukset sanoihin joko `fiery bird` tai `fiery shyly`.[1] Kyselyn tulos on kuvattu tämän kohdan alussa.

Järjestelmämme pitää olla tarpeeksi fiksu, ettei se näytä samoja rivejä useampaan kertaan.

17.1 Luokkahierarkian määrittely

Pääasiallisin keskittymisaiheemme tässä luvussa on luokkahierarkian rakentaminen, joka edustaa käyttäjän kyselyä. Alkusuunnitelmamme on esittää jokainen kyselyoperaatio yksittäisenä luokkana:

```
NameQuery // Shakespeare
NotQuery  // !Shakespeare
OrQuery   // Shakespeare || Marlowe
AndQuery  // William && Shakespeare
```

Jokaiseen luokkaan määritellään eval()-jäsenfunktio, joka ratkaisee kyselyoperaation, jota se edustaa. Esimerkiksi NameQuery:n eval()-jäsenfunktio palauttaa yksinkertaisesti paikkavektorista sanan rivi- ja sarakenumeron, jossa se esiintyy (katso kohta 6.8). OrQuery-luokan eval()-jäsenfunktion pitää kuitenkin muodostaa sen kahden pakkavektorioperandinsa yhdiste jne.

Täten kysely

```
untamed || fiery
```

muodostuu OrQuery-luokkaoliosta, joka sisältää kaksi NameQuery-oliota operandeinaan. Tämä tukee yksinkertaisia kyselyjä, mutta ongelmia syntyy, kun käsitellään yhdistettyjä kyselyjä kuten seuraavassa:

```
Alice || Emma && Weeks
```

Tämä kysely muodostuu kahdesta alikyselystä: OrQuery-oliosta, joka sisältää Alice- ja Emma-NameQuery-oliot sekä AndQuery-oliosta. AndQuery-olion oikeanpuoleinen operandi on NameQuery-olio Weeks.

```
AndQuery
  OrQuery
    NameQuery ("Alice")
    NameQuery ("Emma")
  NameQuery ("Weeks")
```

Vasemmanpuoleinen operandi on kuitenkin OrQuery-olio, joka on sitä ennen. Se voisi esittää yhtä helposti NotQuery- tai muuta AndQuery-oliota yhtä hyvin. Kuinka voimme esittää sisäisesti operandin, kun se voi olla jokin neljästä mahdollisesta kyselyluokkatyypeistä? Ongelmamme on kaksinkertainen:

1. Muista, että voidaksemme yksinkertaistaa toteutustamme, vaadimme, että tyhjä merkki erottaa sanat toisistaan mukaan lukien sulut ja kyselyoperaattorit. Vaikka tämä onkin kohtuuton vaatimus todellisille järjestelmille, se on hyväksyttävä esittelyjaksoissa kuten tässä.

1. Operandin tyyppi pitää kyetä esittelemään OrQuery-, AndQuery- ja NotQuery-luokissa niin, että jokainen voi sisältää minkä tahansa neljästä kyselyluokkatyypistä.
2. Luokkakohtainen ilmentymä pitää kyetä käynnistämään eval()-jäsenfunktiosta suorituksen aikana jokaiselle operandille, päädymme sitten mihin ratkaisuun tahansa 1-kohdan menettelyn kanssa.

Muunlainen kuin oliokeskeinen ratkaisu on määritellä operandin tyypiksi *yhdiste* (*union*) ja tehdä *erottelija*, jolla ilmaistaan operandin todellinen tyyppi:

```
// muu kuin oliosuuntautunut ratkaisu
union op_type {
    // yhdiste ei voi sisältää luokkaoliota
    // niihin liittyvine muodostajineen
    NotQuery *nq;
    OrQuery  *oq;
    AndQuery *aq;
    string   *word;
};

enum opTypes {
    Not_query=1, Or_query, And_query, Name_query
};

class AndQuery {
public:
    // ...
private:
    /*
     * op_types sisältää kyselyn todelliset operandit
     * opTypes yksilöi jokaisen operandin tyypin
     */

    op_type _lop, _rop;
    opTypes _lop_type, _rop_type;
};
```

Vaihtoehtoisesti voimme hylätä yhdisteen ja tallentaa oliot void*-osoittimen kautta kuten seuraavassa:

```
class AndQuery {
public:
    // ...
private:
    void *_lop, *_rop;
    opTypes _lop_type, _rop_type;
};
```

Tarvitsemme silti erottelijan, koska emme voi käsitellä oliota, jota osoitetaan void*-osoittimella suoraan, eikä ole olemassa tapaa kysellä itse osoittimelta sen todellista tyyppiä. (Emme suosittele tätä ratkaisua C++:ssa; se on kuitenkin yleinen ohjelmointiratkaisu C-kielessä.)

Pääasiallisin huono puoli näissä ratkaisuissa on, että tyypin ratkaisu jätetään ohjelmoijan vastuulle. Esimerkiksi void*-ratkaisussa AndQuery:n eval()-operaatio toteutetaan todennäköisesti seuraavasti:

```
void
AndQuery::
eval()
{
    // muu kuin oliosuuntautunut ratkaisu
    // tyypin ratkaisutaakka jää ohjelmoijalle

    // selvitä vasemmanpuoleisen operandin todellinen tyyppi
 switch( _lop_type ) {
     case And_query:
         AndQuery *paq = static_cast<AndQuery*>(_lop);
         paq->eval();
         break;
     case Or_query:
         OrQuery *poq = static_cast<OrQuery*>(_lop);
         poq->eval();
         break;
     case Not_query:
         NotQuery *pnotq = static_cast<NotQuery*>(_lop);
         pnotq->eval();
         break;
     case Name_query:
         NameQuery *pnmq = static_cast<NameQuery*>(_lop);
         pnmq->eval();
         break;
 }

 // sama oikeanpuoleiselle operandille ...
}
```

Kaksi päähaittaa ohjelmoijan tekemässä eksplisiittisessä tyyppiratkaisussa ovat koodin kasvanut koko ja sen monimutkaistuminen, kun käsitellään jokaista tyyppiä suoraan, ja hankaluus lisätä ja poistaa tuettuja tyyppejä rikkomatta olemassaolevaa koodia.

Oliokeskeinen ohjelmointitapa tarjoaa vaihtoehtoisen ratkaisun, jossa tyyppiratkaisu siirretään ohjelmoijalta kääntäjälle. Esimerkiksi seuraavassa on AndQuery:n eval()-operaatio toteutettu uudelleen oliokeskeisellä suunnittelutavalla (eval() on esitelty virtuaaliseksi funktioksi):

```
// oliokeskeinen ratkaisu
// tyyppiratkaisun taakka siirretty kääntäjälle

// huomaa: _lop ja _rop ovat nyt luokkatyyppisiä olioita
// niiden määrittelyt näytetään myöhemmin

void
AndQuery::
eval()
{
  _lop->eval();
  _rop->eval();
}
```

Jos pitäisi lisätä tai poistaa tuettuja tyyppejä, ei tätä koodiosaa tarvitsisi muokata eikä kääntää uudelleen.

17.1.1 Oliokeskeinen suunnittelu

Mistä muodostuu neljän kyselytyypin oliokeskeinen suunnittelu? Kuinka ratkaisemme kaksi aikaisemmin esitettyä ongelmaa?

Määrittelemme *periytymisen* kautta suhteet aikaisemmin riippumattomien kyselyluokkatyyppien välille. Pääsemme tähän esittelemällä abstraktin Query-luokan, joka toimii *kantaluokkana*, josta muut luokat *johdetaan* (eli muodostetaan). Abstraktin luokan voidaan ajatella olevan epätäydellinen luokka, joka saadaan enemmän tai vähemmän täydelliseksi myöhemmin tehtävillä *luokasta johdetuilla* luokilla — meidän tapauksessamme neljä kyselyluokkatyyppiä ovat: AndQuery, OrQuery, NotQuery ja NameQuery.

Abstrakti Query-kantaluokkamme määrittelee tieto- ja jäsenfunktiot, jotka ovat yhteisiä kaikille kyselytyypeille. Query:stä johdettu luokka kuten AndQuery yrittää määritellä, mikä on yksilöllistä juuri sille kyselytyypille. Esimerkiksi NameQuery on tietty Query-ilmentymä, jonka operandi on aina string-merkkijono. Sanomme NameQuery:ä *johdetuksi luokaksi*. Sanomme, että Query toimii sen *kantaluokkana*. (Sama pätee myös muille kyselyluokkatyypeille.) Johdettu luokka perii kantaluokkansa tietojäsenet sekä jäsenfunktiot ja voi käyttää niitä suoraan aivan kuin ne olisivat johdetun luokan jäseniä.

Pääetu periytymishierarkiassa on, että voimme ohjelmoida abstraktin kantaluokan julkista rajapintaa sen sijaan, että ohjelmoisimme yksityisiä tyyppejä, jotka muodostavat sen periytymishierarkian, ja siten suojaamme koodimme hierarkian muutoksilta. Määrittelemme esimerkiksi `eval()`:in abstraktin Query-kantaluokan julkiseksi virtuaalifunktioksi. Kirjoittamalla koodia kuten

```
_rop->eval();
```

on käyttäjän koodi suojattu kyselykielemme vaihtelulta ja muutoksilta. Tämä ei ainoastaan mahdollista tyyppien lisäystä, uudistamista tai poistamista ilman vaatimusta muuttaa käyttäjien ohjelmia, vaan vapauttaa uuden kyselytyypin tekijän koodaamasta uudelleen itse hier-

arkian kaikkien tyyppien yhteisiä toimintoja ja käyttäytymisiä. Tätä tukee kaksi erityistä periytymispiirrettä: *monimuotoisuus* ja *dynaaminen sidonta*.

Kun puhumme C++:n monimuotoisuudesta, tarkoitamme pääasiassa kantaluokan osoittimen tai viittauksen kykyä osoittaa kaikkiin siitä johdettuihin luokkiin. Jos esimerkiksi määrittelemme jäsenettömän eval()-funktion seuraavasti

```
// pquery voi osoittaa mihin tahansa Query:stä johdettuun luokkaan
void eval( const Query *pquery )
{
    pquery->eval();
}
```

voimme käynnistää sen kelvollisesti välittämällä sille minkä tahansa neljän kyselytyypin olion osoitteen:

```
int main()
{
  AndQuery aq;
  NotQuery notq;
  OrQuery *oq = new OrQuery;
  NameQuery nq( "Botticelli" );

  // ok: jokainen on johdettu Query:stä
  // kääntäjä konvertoi kantaluokaksi automaattisesti

  eval( &aq );
  eval( &notq );
  eval( oq );
  eval( &nq );
}
```

kun taas yritys käynnistää eval() sellaisen olion osoitteella, jota ei ole johdettu Query:stä, johtaa käännösvirheeseen:

```
int main()
{
    string name( "Scooby-Doo" );

    // virhe: string:iä ei ole johdettu Query:stä
    eval( &name );
}
```

Kun eval():issa suoritetaan

```
pquery->eval();
```

pitää sen käynnistää sopiva eval()-virtuaalijäsenfunktio, joka perustuu todelliseen luokkaolioon, jota pquery osoittaa. Edellisessä esimerkissä pquery osoittaa vuorollaan AndQuery-, NotQuery-, OrQuery- ja NameQuery-oliota. Ohjelman suorituksen aikana jokaisen käynnistyskohdassa päätellään todellinen luokkatyyppi, jota pquery osoittaa, ja vastaavaa eval()-

ilmentymää kutsutaan. *Dynaaminen sitominen* on mekanismi, jonka avulla tämä toteutetaan. (Katsomme virtuaalifunktioiden suunnittelua ja käyttöä tarkemmin kohdassa 17.5.)

Oliokeskeisessä paradigmassa ohjelmoija käsittelee tuntematonta tyyppi-ilmentymäjoukkoa, joka on sidottu, mutta loputon. (Tyyppijoukko on sidottu sen periytymishierarkiaan. Teoriassa ei ole rajoja tuon hierarkian leveydelle eikä syvyydelle.) C++:ssa tämä saavutetaan käsittelemällä olioita vain kantaluokan osoittimilla ja viittauksilla. Oliopohjaisessa paradigmassa ohjelmoija käsittelee ilmentymää, joka on kiinteä, erityinen tyyppi ja täydellisesti määritelty käännöshetkellä.

Vaikka olion monimuotoinen käsittely vaatii, että oliota käsitellään osoittimen tai viittauksen kautta, ei osoittimen tai viittauksen käsittely itsessään johda C++:ssa välttämättä monimuotoisuuteen. Mietitäänpä esimerkiksi tätä:

```
// ei monimuotoisuutta
int *pi;

// ei kielen tukemaa monimuotoisuutta
void *pvi;

// ok: pquery voi osoittaa mihin tahansa Query:n johdannaiseen
Query *pquery;
```

C++:ssa monimuotoisuus esiintyy vain yksittäisissä luokkahierarkioissa. Osoitintyyppejä kuten `void*` voidaan kuvata monimuotoisina, mutta ne ovat ilman kielen eksplisiittistä tukea — ohjelmoijan pitää käsitellä niitä eksplisiittisten tyyppimuunnosten kautta ja jonkinlaisten erottelijoiden avulla, joilla pidetään kirjaa osoitetuista todellisista tyypeistä. (Voidaan sanoa, että ne eivät ole ensimmäisen luokan monimuotoisia olioita.)

C++-kieli tukee monimuotoisuutta seuraavilla tavoilla:

1. Implisiittisen konversion kautta, joka kohdistuu johdetun luokan osoittimeen, viittaukseen osoittimeen tai viittaukseen sen julkiseen kantatyyppiin:

   ```
   Query *pquery = new NameQuery( "Glass" );
   ```

2. Virtuaalifunktiomekanismin kautta:

   ```
   pquery->eval();
   ```

3. Operaattorien `dynamic_cast` ja `typeid` kautta (nämä käsitellään tarkemmin kohdassa 19.1):

   ```
   if ( NameQuery *pnq =
      dynamic_cast< NameQuery* >( pquery )) ...
   ```

Ratkaisemme ongelmamme määrittelemällä jokaisen And-, Query-, NotQuery- ja OrQuery-luokan operandin `Query*`-tyyppisenä osoittimena. Esimerkiksi:

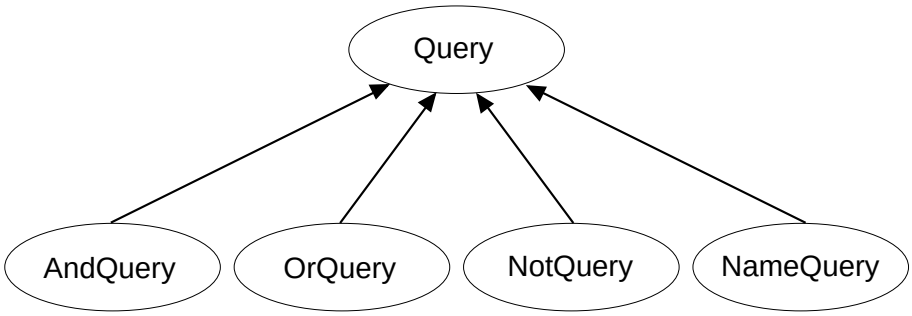
```
class AndQuery {
public:
  // ...
```

```
private:
   Query *_lop;
   Query *_rop;
};
```

Molemmat operandit voivat nyt osoittaa mitä tahansa kyselyluokan tyyppiä, joka on johdettu abstraktista kantaluokasta joko nyt tai tulevaisuudessa. Jokaisen operandin arviointi, joka tapahtuu ohjelman suorituksen aikana, on riippumaton todellisesta tyypistään virtuaalimekanismin ansiosta:

```
_rop->eval();
```

Kuvassa 17.1 kuvataan abstraktin Query-luokan periytymishierarkiaa ja sen neljää johdettua luokkaa. Kuinka muunnamme kuvan 17.1 C++-ohjelmakoodiksi?



Kuva 17.1 Query-luokkahierarkia

Kohdassa 2.4 kävimme läpi IntArray-luokkahierarkian toteutuksen. Kuvaan 17.1 määritelty Query-luokkahierarkia on syntaktisesti samanlainen:

```
class Query { ... };
class AndQuery  : public Query { ... };
class OrQuery   : public Query { ... };
class NotQuery  : public Query { ... };
class NameQuery : public Query { ... };
```

Periytyminen määritetään *luokasta johdettujen luettelolla* (*class derivation list*). Yksiperiytymisessä sen yleinen muoto on seuraava

```
: access-level base-class
```

jossa access-level on joko public, protected tai private (protected- ja private-periytymisen merkitystä käsitellään kohdassa 18.3) ja base-class on edellä määritellyn luokan nimi. Esimerkiksi Query toimii neljän kyselytyypin julkisena kantaluokkana.

Johdettujen luokkien luettelossa oleva luokka pitää olla määritelty ennen kuin se määritetään kantaluokaksi. Esimerkiksi seuraava Query:n jatkoesittely ei riitä sille, jotta se voisi toimia kantaluokkana:

```
// virhe: Query pitää määritellä
class Query;
class NameQuery : public Query { ... };
```

Johdetun luokan jatkoesittelyyn ei kuulu johdettujen luokkien luetteloa, vaan yksinkertaisesti sen luokan nimi — aivan samoin kuin se olisi johtamaton luokka. Esimerkiksi seuraava NameQuery:n jatkoesittely johtaa käännösvirheeseen:

```
// virhe: johdetun luokan jatkoesittelyyn ei pidä ottaa
// mukaan johdettujen luokkien luetteloa
class NameQuery : public Query;
```

Virheetön jatkoesittely on seuraavanlainen:

```
// sekä johdettujen että johtamattomien luokkien
// jatkoesittelyihin kuuluu vain luokan nimi
class Query;
class NameQuery;
```

Pääero Query-kantaluokan ja kohdan 2.4 IntArray-kantaluokan välillä on, että Query ei edusta varsinaista oliota sovellusalueellamme. IntArray-luokkahierarkian käyttäjät määrittelevät ja käsittelevät todennäköisesti IntArray-luokkaolioita suoraan. Query-luokkahierarkian käyttäjät määrittelevät vain Query-osoittimia ja -viittauksia. Näitä käytetään Query:stä johdettujen luokkatyyppisten olioiden epäsuoraan käsittelyyn. Query:n sanotaan olevan *abstrakti kantaluokka*; IntArray on *konkreettinen kantaluokka*. Oliokeskeisen suunnittelun vallitseva tapa on abstraktin kantaluokan kuten Query:n ja yhden, julkisesti johdetun luokan määrittely.

Harjoitus 17.1

Kirjasto tukee seuraavia lainausmateriaalien kategorioita, joilla jokaisella on oma lainaus- ja palautuspolitiikkansa. Järjestä nämä periytymishierarkiaan:

```
kirja         äänikirja
äänilevy      lasten kirja
video         sega-videopeli
lainakirja    sony playstation -videopeli
cdrom-kirja   nintendo-videopeli
```

Harjoitus 17.2

Valitse yksi seuraavista yleisabstraktioista, jotka sisältävät tyyppiperheitä (tai valitse jokin oma). Järjestä tyypit periytymishierarkiaan:

(1) Graafiset tiedostoformaatit (kuten gif, tiff, jpeg, bmp)
(2) Geometriset primitiivit (kuten neliö, ympyrä, pallo, kartio)
(3) C++-kielen tyypit (kuten luokka, funktio, jäsenfunktio)

17.2 Hierarkian jäsenten tunnistaminen

Kuten kuvasimme kohdassa 2.4, oliopohjaisessa suunnittelussa luokalla on yleensä yksi tekijä ja monta käyttäjää. Tekijä suunnittelee ja usein toteuttaa luokan. Käyttäjät opettelevat käyttämään julkista rajapintaa, jonka tekijä on antanut käytettäväksi. Tämä toimintojen erottelu vaikuttaa luokan jakamiseen private- ja public-käsittelytasoihin.

Periytymisessä on nyt useita luokan tekijöitä: yksi tekee kantaluokan toteutuksen (ja mahdollisesti useita johdettuja luokkia) ja yksi tai useammat tekevät johdetut luokat periytymishierarkian elinkaaren aikana. Tämä toiminta on myös toteutustoimintaa. Alityypin tekijä usein (mutta ei aina) tarvitsee pääsyn kantaluokan toteutukseen. Jotta sen voisi tehdä ja silti estää yleisen pääsyn toteutukseen, on tehty lisäkäsittelytaso, protected. Vaikka luokan protected-osan tietojäsenet ja jäsenfunktiot ovat yleisohjelman saavuttamattomissa, ne ovat silti johdetun luokan käytettävissä. (Kaikki kantaluokan private-osaan sijoitettu on vain luokan käytettävissä eikä yhdenkään johdetun luokan.)

Kriteeri jäsenen julkiseksi asettamiselle luokassa ei vaihtele oliopohjaisen ja oliokeskeisen suunnittelun välillä. Se mikä vaihtelee, on se, esitelläkö ei-julkisia jäseniä suojattuina vai yksityisinä. Luokan jäsenestä tehdään yksityinen (private), jos halutaan estää jatkossa johdettujen luokkien suora pääsy tuohon jäseneen. Jäsenestä tehdään suojattu (protected), jos uskotaan, että se tekee operaation tai tiedon tallennuksen, johon siitä myöhemmin johdettu luokka vaatii suoran pääsyn tehokkaan toteutuksen vuoksi. Suunnittelun lisänäkökohta luokalle, jonka suunnitellaan toimivan kantaluokkana, on tyyppiriippuvaisten jäsenfunktioiden tunnistaminen. Ne ovat luokkahierarkian virtuaalifunktioita.

Seuraava vaihe Query-luokkahierarkian suunnittelussamme on päättää seuraavaa:

1. Mitä operaatioita tulisi Query-luokkahierarkian julkisen rajapinnan tehdä?
2. Mitkä näistä tulisi esitellä virtuaalisiksi?
3. Mitä lisäoperaatioita, jos yhtäkään, yksittäiset johdetut luokkatyypit vaativat?
4. Mitä tietojäseniä, jos yhtäkään, tulisi esitellä abstraktiin Query-luokkaamme?
5. Mitä tietojäseniä, jos yhtäkään, yksittäiset johdetut luokkatyypit vaativat?

Valitettavasti ei ole olemassa taikakaavaa näiden kysymysten vastauksiin. Ja vaikka vastaukset saadaankin, ei ole takuita niiden oikeellisuudesta tai täydellisyydestä. Kuten tulemme näkemään, oliokeskeisen suunnittelun prosessi on iteratiivinen ja vaatii sekä lisäyksiä että

muokkauksia kehittyvään luokkahierarkiaan. Tämän kohdan 17.2 loppuun mennessä olemme käyneet läpi Query-luokkahierarkiamme iteratiivisen kehityksen.

17.2.1 Kantaluokan määrittely

Query-luokan jäsenet edustavat

1. Operaatioita, joita tukevat kaikki johdetut kyselyluokkatyypit. Tähän kuuluvat sekä virtuaalioperaatiot, jotka johdetut luokkatyypit korvaavat, ja ei-virtuaaliset operaatiot, jotka johdetut luokat jakavat keskenään. Katsomme esimerkin jokaisesta.
2. Tietojäseniä, jotka ovat yhteisiä johdetuille luokille. Erottelemalla nämä jäsenet johdetuista luokista abstraktiin Query-luokkaamme, kykenemme käsittelemään jäseniä riippumattomasti todellisesta tyypistä, jota olemme käsittelemässä. Jälleen: katsomme kahta esimerkkiä.

Jos on tehty kysely kuten

```
fiery || untamed
```

ovat kaksi pääoperaatiota (1) kyselyä vastaavien tekstirivien arviointi (2) täsmäävien rivien näyttäminen käyttäjälle. Nimeämme nämä operaatiot vastaavasti näin: eval() ja display().

eval()-operaation suoritus on erityinen jokaiselle johdetulle kyselyluokkatyypille ja pitää siksi esitellä virtuaaliseksi Query-luokan määrittelyyn. Jokaisen johdetun luokan pitää tehdä oma toteutuksensa. Query-kantaluokassa on julkinen rajapinta, jota ohjelmoimme.

Täsmäävien rivien näyttäminen display()-operaatiolla on riippumaton todellisen johdetun kyselyluokan tyypistä. Algoritmi vaatii pääsyn itse tekstin esitystapaan ja riviluetteloon, joka täsmää kyselyyn. Tämä algoritmi on muuttumaton, oli operaatio AndQuery, OrQuery tai tms. Sen sijaan, että monistaisimme operaatiota ja tukisimme joka johdetussa luokassa olevaa tietoa, määrittelemme yhden ilmentymän Query:yn, jonka jokainen on perinyt.

Tällä suunnitelmalla voimme käynnistää molemmat operaatiot tietämättä olion tyyppiä, jota olemme käsittelemässä. Esimerkiksi:

```
void
doit( Query *pq )
{
    // virtuaalikäynnistys
    pq->eval();

    // Query::display():n staattinen käynnistys
    pq->display();
}
```

Kuinka täsmäävät tekstirivit tulisi esittää? Kyselyn jokaisen sanan mukanaolo ilmaistaan paikkavektorilla, joka rakennetaan tekstin käsittelyn aikana. Muista, että paikka on kokonaislukupari, rivi ja sarake. Sanojen sijaintikartta, jonka build_text_map() muodostaa, sisältää järjestelmän havaitsemien kaikkien sanojen kaikki paikkavektorit. Karttaa indeksoidaan merkkijonolla, joka edustaa itse sanaa. Jos syöteteksti on esimerkiksi

```
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?"
```

niin seuraavassa on tekstin sijaintikartan rivit joillekin sanoille useine esiintymisineen (sana edustaa avainta karttaan; suluissa olevat arvoparit ovat paikkavektorin elementtejä — huomaa, että rivit ja sarakkeet on numeroitu alkaen arvosta 0):

```
bird ((2,3),(2,9))
daddy ((0,8),(3,3),(5,5))
fiery ((2,2),(2,8))
hair ((0,6),(1,6))
her ((0,7),(1,5),(2,12),(4,11))
him ((4,2),(4,8))
she ((4,0),(5,1))
tell ((2,11),(4,1),(4,10))
```

Paikkavektori ei kuitenkaan välttämättä edusta kyselyn ratkaisua. Esimerkiksi fiery esiintyy kahdessa paikassa, mutta näkyy vain yhdellä rivillä näytöllä.

Meidän pitää laskea paikkavektorista samanlaiset rivit. Eräs strategia on luoda vektori jokaisesta paikkavektorin rivinumerosta; välitä rivivektori geneeriselle unique()-algoritmille, joka poistaa tuplaelementit (katso unique():n esittely ja esimerkki kirjan liitteestä). Jäljelle jäävien rivien pitäisi jo olla nousevassa järjestyksessä. Jotta voisimme olla siitä varmoja, voimme käyttää geneeristä sort()-algoritmia rivivektoriin.

Vaihtoehtoinen strategia, jonka olemme valinneet, on rakentaa joukko-olio paikkavektorin rivinumeroista. Joukko-olio pitää automaattisesti yksilöllisiä arvoja nousevassa järjestyksessä. Tarvitsemme funktion, jolla käännämme paikkavektorimme yksilölliset rivinumerot joukoksi:

```
set<short>* Query::_vec2set( const vector< location >* );
```

Esittelemme _vec2set():in suojatuksi Query-jäsenfunktioksi. Se ei ole julkinen, koska se ei kuulu operaatioiden joukkoon, joita haluamme Query-luokkahierarkiamme käyttäjien käynnistävän. Se ei ole yksityinen, koska se on apufunktio, jonka haluamme saada johdettujen luokkien käytettäväksi. (Alaviivan tarkoitus on ilmaista, että se ei ole osa Query-luokkahierarkian julkista rajapintaa.)

Esimerkiksi paikkavektori sisältää bird-sanalle kaksi tietoa samalla rivillä. Sen ratkaisujoukko muodostuu siten yhdestä tiedosta: (2). Paikkavektori sisältää tell-sanalle kolme tietoa, kaksi samalla rivillä. Sen ratkaisujoukko on siten kaksi tietoa: (2,4). Tässä ovat vastaavat ratkaisujoukot aikaisemmin listatuille paikkavektoreille:

```
bird   (2)
daddy  (0,3,5)
fiery  (2)
hair   (0,1)
her    (0,1,2,4)
```

```
him    (4)
she    (4,5)
tell   (2,4)
```

NameQuery ratkaistaan yksinkertaisesti hakemalla sen nimeen liittyvä paikkavektori, kääntämällä tuo vektori yksilöllisten rivien joukoksi ja näyttämällä vastaavat tekstirivit.

NotQuery edustaa kaikkia rivejä, joissa sen operandi ei esiinny. Kysely

```
! daddy
```

edustaa rivijoukkoa (1,2,4). Jotta tämä voitaisiin ratkaista, pitää tietää, kuinka monta riviä tekstiin sisältyy. (Emme koskaan laskeneet tätä tietoa, koska se ei ole tullut mieleemme kuin vasta nyt, kun tarvitsemme sitä; kuitenkin pian tulee selväksi, että tarvitsemme jopa enemmän tietoa!) Jotta voisimme helpommin ratkaista NotQuery-operaation, on kätevää, jos generoimme joukon kaikista tekstin riveistä (0,1,2,3,4,5). Sitten voimme ratkaista tämän ottamalla erotusjoukon (set_difference()) näistä kahdesta joukosta. (NameQuery:n daddy-joukko on (0,3,5).)

OrQuery-edustaa kaikkien niiden rivien yhdistettä, joissa jokainen sen operandi esiintyy. Esimerkiksi kyselyssä

```
fiery || her
```

on yksilöllisten rivien joukko (0,1,2,4). Tämä on fiery-sanaan liittyvän rivin (2) ja her-sanaan liittyvien rivien (0,1,2,4) yhdiste. Riviratkaisumme ei saa sisältää tupla-arvoja ja sen pitää olla nousevassa järjestyksessä.

Tähän saakka olemme kyenneet ratkaisemaan jokaisen kyselyn käsittelemällä yksinkertaisesti yksittäisiä riviesiintymiä. AndQuery kuitenkin vaatii, että tutkimme sijaintiparin sekä rivi- että sarakearvot. Esimerkiksi kyselyn operandit

```
her && hair
```

esiintyvät neljällä eri rivillä. AndQuery:n ajatus kuitenkin vaatii, kuten olemme sen määritelleet, että täsmäävä rivi sisältää täsmälleen merkkijonon her hair. Ensimmäisen rivin esiintymä ei täsmää, vaikka sanat ovat vierekkäin

```
Alice Emma has long flowing red hair. Her Daddy says
```

kun taas toisen rivin kahden sanan esiintymä täsmää täysin:

```
when the wind blows through her hair, it looks almost alive,
```

her-sanan kaksi muuta esiintymää eivät ole hair-sanan vieressä — ne eivät selvästikään täsmää. Kyselyn ratkaisu on siten tekstin toinen rivi: (1).

Jos AndQuery-operaatiota ei olisi, meidän ei tarvitsisi ylläpitää paikkavektoria jokaiselle operaatiolle. Koska kuitenkin jokaisella johdetulla kyselytyypillä voi olla AndQuery-operandi, pitää jokaisen laskea ja ylläpitää, ei vain yksilöllisiä rivjeä, vaan myös rivi- ja sarakesijaintipari. Mietitäänpä esimerkiksi kumpaakin seuraavaa kahta kyselyä:

```
fiery && ( hair || bird || potato )
fiery && ( ! burr )
```

Mahdollisuus, että NotQuery olisi AndQuery:n operandi, tarkoittaa se sitä, että meidän pitää luoda, ei vain yksinkertaisesti vektori, joka sisältää jokaisen rivinumeron tiedon, vaan vektori, joka sisältää jokaisen rivi- ja sarakeparin sijainnin tekstissä. (Katsomme tätä jälleen, kun käymme läpi NotQuery:n eval()-funktiota kohdassa 17.5.)

Eräs välttämätön tietojäsen on siten paikkavektori, joka liittyy jokaisen operaation arviointiin. Meillä on vapaus esitellä se jokaisen johdetun luokan jäseneksi tai abstraktin Query-kantaluokan jäseneksi, jonka jokainen johdettu luokka perii. Jäsenelle vaadittava tila on sama molemmissa ratkaisuissa. Kun sijoitamme sen yhteiseen Query-kantaluokkaan, lokalisoimme sen alustamisen ja käsittelytuen. Tämä on se, minkä olemme päättäneet tehdä.

Se, valitsemmeko yksilöllisten rivinumeroiden esitykseen (kutsumme sitä *riviratkaisujoukoksi*) tietojäsenen vai laskemmeko sen “lennossa” joka kerta, kun tarvitsemme sitä, on toteutuksen päätettävissä. Olemme päättäneet laskea sen vaadittaessa ja sitten laittaa syrjään sen osoitteen myöhempää käsittelyä varten. Se myös esitellään abstraktin Query-kantaluokan jäsenenä.

Jotta voisimme näyttää täsmäävät rivit, tarvitsemme sekä riviratkaisujoukon että varsinaisen tekstitiedoston, johon rivit on tallennettu. Kun taas jokainen operaatio vaatii kuitenkin oman paikkavektorin, on vain yksi, jaettu tekstitiedostoilmentymä tarpeen. Tästä syystä määrittelemme sen Query:n staattiseksi tietojäseneksi. (display()-funktion toteutus riippuu vain näistä kahdesta jäsenestä.)

Tässä on sitten ensimmäinen iteraatioesitys abstraktista Query-kantaluokastamme, mutta ilman muodostajien, tuhoajien tai kopioinnin sijoitusoperaattoreiden esittelyjä (palaamme näihin vastaavasti kohdissa 17.4 ja 17.6):

```
#include <vector>
#include <set>
#include <string>
#include <utility>

typedef pair< short, short > location;

class Query {
public:
   // muodostajia ja tuhoajia
   // käsitellään kohdassa 17.4

      // kopiointimuodostajaa ja kopioinnin sijoitusoperaattoria
      // käsitellään kohdassa 17.6
```

```
    // operaatiot, jotka tukevat julkista rajapintaa
    virtual void eval() = 0;
    void display () const;

    // lukukäsittelyfunktiot
    const set<short> *solution() const;
    const vector<location> *locations() const { return &_loc; }

    static const vector<string> *text_file() {return _text_file;}

protected:
    set<short> *_vec2set( const vector<location>* );

    static vector<string> *_text_file;

    set<short>      *_solution;
    vector<location>  _loc;
};

inline const set<short>*
Query::
solution()
{
    return _solution
            ? _solution
            : _solution = _vec2set( &_loc );
}
```

Erikoinen syntaksi

```
virtual void eval() = 0;
```

ilmaisee, että abstraktin Query-kantaluokan `eval()`-funktiolle ei ole tehty virtuaalimäärittelyä. Miksi? Koska ei ole olemassa järkevää algoritmia sen määrittelemiseen. `eval()`-ilmentymää kutsutaan *puhtaaksi virtuaalifunktioksi* (*pure virtual function*). Se toimii kuin paikanpitäjänä luokkahierarkian julkisessa rajapinnassa. Sitä ei aiota käynnistää koskaan ohjelmassamme. Sen sijaan jatkossa jokainen johdettu luokka tekee todellisen ilmentymän. (Katsomme virtuaalifunktioita tarkemmin kohdassa 17.5.)

17.2.2 Johdettujen luokkien määrittely

Jokainen johdettu luokka perii kantaluokkansa tietojäsenet ja jäsenfunktiot. Johdettuun luokkaan tarvitsee ohjelmoida vain ne piirteet, jotka eroavat kantaluokasta tai laajentavat kantaluokan käyttäytymistä. Esimerkiksi NameQuery:yn pitää määritellä eval()-ilmentymä. Sen lisäksi sen pitää tukea sanan nimeä. Esitämme nimen string-jäsenluokkaoliolla. Lopuksi, jotta voisimme hakea vastaavan paikkavektorin, pitää sanojen sijaintikartan olla käytettävissä. Koska tarvitaan vain yksi ilmentymä, jonka kaikki NameQuery-luokkaoliot jakavat, esittelemme sen staattiseksi tietojäseneksi. Tässä on aluksi NameQuery-luokkamäärittely (jälleen huomauttaen, että viivytämme muodostajien, tuhoajan ja kopioinnin sijoitusoperaattorin käsittelyä vielä):

```
typedef vector<location> loc;

class NameQuery : public Query {
public:
   // ...

   // korvaa virtuaalisen Query::eval()-ilmentymän[2]
   void eval();

   // lukukäsittelyfunktio
   string name() const { return _name; }

   static const map<string,loc*> *word_map() { return _word_map; }

protected:
   string _name;
   static map<string,loc*> *_word_map;
};
```

Sen lisäksi, että NotQuery-luokkaan tehdään sen oma ilmentymä eval()-virtuaalifunktiosta, siihen pitää tehdä tuki sen yhdelle operandille. Koska operandi voi olla mikä tahansa johdetuista kyselyluokkatyypeistä, määrittelemme sen osoitintyypiksi Query:yn. Muista, että NotQuery-ratkaisun pitää edustaa, ei vain tekstirivejä, joissa sen operandi ei esiinny, vaan myös jokaisen rivin sarakepaikkoja. Esimerkiksi kyselyssä NotQuery

```
! daddy
```

sen NameQuery-operandi sisältää seuraavan paikkavektorin:

```
daddy ((0,8),(3,3),(5,5))
```

NotQuery:n paikkavektorin pitää sisältää rivien (1,2,4) jokainen sarake. Lisäksi sen pitää sisältää rivin (0) jokainen sarake, paitsi sarake (8), rivin (3) jokainen sarake, paitsi sarake (3) ja

2. Johdetun luokan perityn virtuaalifunktion ilmentymää kuten eval() ei tarvitse, mutta voidaan määrittää virtual-avainsanalla. Kääntäjä havaitsee ilmentymän funktion prototyypin vertailun perusteella.

rivin (5) jokainen sarake, paitsi sarake (5).

Yksinkertaisin tapa tämän laskemiseen on yksi jaettu paikkavektori, joka sisältää rivi- ja sarakeparin jokaiselle sanan esiintymälle tekstissä; katsomme toteutusta, kun esittelemme NotQuery:n eval()-funktion kohdassa 17.5. Joka tapauksessa määrittelemme tämän jäsenen NotQuery-luokan staattiseksi tietojäseneksi.

Tässä on aluksi NotQuery-luokan määrittely (jälleen kerran huomauttaen, että viivytämme vielä muodostajien, tuhoajan ja kopioinnin sijoitusoperaattorin esittelyä):

```
class NotQuery : public Query {
public:
   // ...

   // vaihtoehtoinen syntaksi: eksplisiittinen virtual-avainsanan
   // ilmentymä korvaa Query::eval():in
   virtual void eval();

   // lukukäsittelyfunktiot
   const Query *op() const { return _op; }
   static const vector<location>*all_locs(){ return _all_locs; }

protected:
   Query *_op;
   static const vector< location > *_all_locs;
};
```

AndQuery- ja OrQuery-luokat ovat molemmat binäärioperaatioita ja siksi pitää tukea oikeanpuoleista ja vasemmanpuoleista operandia. Molemmat operandit voivat olla mitä tahansa johdetuista kyselyluokkatyypeistä, joten määrittelemme molemmat jäsenet osoitintyypeiksi Query:yn. Kumpaankin pitää tehdä oma vastaava ilmentymä eval()-virtuaalifunktiosta. Tässä on aluksi OrQuery-luokan määrittely:

```
class OrQuery : public Query {
public:
     // ...

     virtual void eval();

     const Query *rop() const { return _rop; }
     const Query *lop() const { return _lop; }

protected:
     Query *_lop;
     Query *_rop;
};
```

Lisäksi jokaisella AndQuery-luokkaoliolla pitää olla pääsy jokaisen rivin sisältämään sanojen lukumäärään. Muussa tapauksessa AndQuery:n vertailu ei voi löytää vierekkäisiä sanoja, jotka jakautuvat kahdelle riville. Esimerkiksi kyselyssä

```
tell && her && magical
```

täsmäävät sanat jakautuvat kolmannelle ja neljännelle riville:

```
like a fiery bird in flight. A beautiful fiery bird, he tells her,
magical but untamed. "Daddy, shush, there is no such thing,"
```

Kolmeen sanaan liittyvät paikkavektorit ovat seuraavat:

```
her     ((0,7),(1,5),(2,12),(4,11))
magical ((3,0))
tell    ((2,11),(4,1),(4,10))
```

Ellei AndQuery:n eval()-funktio pysty päättelemään, että rivi (2) sisältää 12 sanaa, se ei voi päätellä, että magical on her:in vieressä. Teemme yhden ilmentymän käytettäväksi staattisen tietojäsenen kautta, jonka olemme nimenneet _max_col. (eval()-toteutus on yksityiskohtaisesti kohdassa 17.5.) Tässä on sitten ensimmäinen AndQuery-luokan määrittely:

```
class AndQuery : public Query {
public:
    // muodostajia käsitellään kohdassa 17.4
    virtual void eval();

    const Query *rop() const { return _rop; }
    const Query *lop() const { return _lop; }

    static void max_col( const vector< int > *pcol )
            { if ( !_max_col ) _max_col = pcol; }

protected:
    Query *_lop;
    Query *_rop;
    static const vector< int > *_max_col;
};
```

17.2.3 Yhteenveto

Jokaisen neljän johtamasi luokan julkinen rajapinta muodostuu Query-luokasta perityistä julkisista jäsenistä ja yksittäisten johdettujen luokkien julkisista jäsenistä. Kun kirjoitamme

```
Query *pq = new NameQuery( "Monet" );
```

voimme käsitellä vain Query-luokan julkista rajapintaa pq:n kautta. Kun kirjoitamme

```
pq->eval();
```

ja koska eval() on esitelty virtuaalifunktioksi, käynnistetään johdetun luokan eval()-ilmentymä, jota pq todellisuudessa osoittaa. Tässä tapauksessa käynnistetään NameQuery-ilmentymä. Kun kirjoitamme

```
pq->display();
```

käynnistetään aina Query:n ei-virtuaalinen display()-funktio. Kuitenkin display() yhä vaikuttaa pq:n osoittaman johdetun luokkaolion ratkaisujoukkoon. Sen sijaan, että luottaisimme virtuaalimekanismiin, laitoimme jaetun operaation ja sitä tukevan tiedon yhteiseen abstraktiin Query-kantaluokkaan. display() on esimerkki monimuotoisesta ohjelmoinnista, jota eivät tue virtuaalifunktiot, vaan periytyminen yksin. Tässä on toteutuksemme (tämä on väliaikainen ratkaisu kuten tulemme näkemään viimeisessä kohdassa):

```
void
Query::
display()
{
   if ( ! _solution->size() ) {
        cout << "\n\tSorry, "
             << " no matching lines were found in text.\n"
             << endl;
   }

   set<short>::const_iterator
                    it = _solution->begin(),
                    end_it = _solution->end();

   for ( ; it != end_it; ++it ) {
       int line = *it;

       // älä sekoita käyttäjää tekstiriveillä, jotka alkavat arvosta 0 ...
       cout << "( " << line+1 << " ) "
            << (*_text_file)[line] << '\n';
     }

   cout << endl;
}
```

Tässä kohtaa olemme tehneet ensimmäisen määrittelyn Query-luokkahierarkiastamme. Eräs kysymys, jota emme ole vielä miettineet, on, kuinka aiomme luokkahierarkiaa käyttäen rakentaa varsinaisen tietorakenteen käyttäjän kyselyn esittämiseen. Itse asiassa, jos toteutuksemme tukisi tätä, muokkaisimme ja laajentaisimme määrittelyä, jonka juuri olemme saaneet aikaiseksi. Ennen kuin palaamme siihen, meidän pitää tutkia periytymisen mekanismia C++:ssa tarkemmin.

Harjoitus 17.3

Mietipä seuraavia kohdan 17.1 harjoituksen 17.1 kirjastoluokkahierarkian jäseniä. Yksilöi, mitkä ilmentymät ovat todennäköisesti virtuaalifunktioehdokkaita ja mitkä ovat yhteisiä (vai ovatko mitkään) kaikille kirjastomateriaaleille ja siten voidaan täysin esittää kantaluokassa. (Huomaa: LibMember on abstraktio, joka edustaa kirjaston jäsentä, joka voi lainata kirjastomateriaaleja. Date on luokka, joka edustaa tietyn vuoden kalenteripäivää.)

```
class Library {
public:
    bool check_out( LibMember* );
    bool check_in ( LibMember* );
    bool is_late( const Date& today );
    double apply_fine();
    ostream& print( ostream&=cout );

    Date* due_date() const;
    Date* date_borrowed() const;

    string title() const;
    const LibMember* member() const;
};
```

Harjoitus 17.4

Yksilöi kohdan 17.1 harjoituksessa 17.2 valitun luokkahierarkian kantaluokan ja johdetun luokan jäsenet. Yksilöi virtuaalifunktiot kuten myös julkiset ja suojatut jäsenet.

Harjoitus 17.5

Mitkä seuraavista ovat virheellisiä, vai onko yksikään?

```
class Base { ... };

(a) class Derived : public Derived { ... };
(b) class Derived : Base { ... };
(c) class Derived : private Base { ... };
(d) class Derived : public Base;
(e) class Derived inherits Base { ... };
```

17.3 Kantaluokan jäsenten käsittely

Johdettu luokkaolio muodostuu todellisuudessa useista osista. Jokainen kantaluokka edustaa *alioliota*, joka muodostuu kantaluokan ei-staattisista tietojäsenistä. Johdettu luokkaolio muodostuu kantaluokkansa aliolioista ja johdetusta osasta, joka muodostuu johdetun luokan ei-staattisista tietojäsenistä. Esimerkiksi NameQuery-oliomme muodostuu Query-alioliosta, joka sisältää perityt _loc- ja _solution-tietojäsenet ja NameQuery-luokkaosan, joka sisältää _name-tietojäsenen.

Johdetussa luokassa voidaan käsitellä kantaluokka-aliolion jäseniä suoraan aivan kuin ne olisivat johdetun luokan jäseniä. (Periytymisketjun syvyys ei rajoita näiden jäsenien käsittelyä eikä se kasvata tuon käsittelyn kuormaa.) Esimerkiksi:

```
void
NameQuery::
display_partial_solution( ostream &os )
{
```

```
    os << _name
       << " is found in "
       << (_solution ? _solution->size() : 0)
       << " lines of text\n";
}
```

Sama pätee kantaluokasta perittyjen jäsenfunktioiden käsittelyyn: käynnistämme niitä aivan kuin ne olisivat johdetun luokan jäseniä, joko luokan olion kautta

```
NameQuery nq( "Frost" );

// käynnistää NameQuery::eval():in
nq.eval();

// käynnistää Query::display():n
nq.display();
```

tai suoraan jäsenfunktiossa

```
void
NameQuery::
match_count()
{
    if ( ! _solution )
        // käynnistää Query::_vec2set():in
        _solution = _vec2set( &_loc );
    return _solution->size();
}
```

On olemassa yksi poikkeus kantaluokan jäsenen suoraan käsittelyyn johdetussa luokassa — kun kantaluokan jäsenen nimeä käytetään uudelleen johdetussa luokassa. Esimerkiksi:

```
class Diffident {
public: // ...
protected:
   int _mumble;
   // ...
};

class Shy : public Diffident {
public: // ...
protected:
   // sananmukaisesti peittää Diffident::_mumble:n näkyvyyden
   string _mumble;

   // ...
};
```

Shy:n viittausalueella _mumble:n tarkentamaton käyttö ratkaistaan aina Shy:n _mumble-string-jäseneksi, vaikka sen käyttö ei ole sallittu. Esimerkiksi:

```
void
Shy::
turn_eyes_down()
{
   // ...
   _mumble = "excuse me"; // ok

   // virhe: int Diffident::_mumble peittyy
   _mumble = -1;
}
```

Ei ole epätavallista kuulla ohjelmoijien mutisevan kääntäjänsä typeryyttä, kun kääntäjä antaa virheilmoituksen tyyppivirheestä tämäntyyppisessä käyttötilanteessa. Vaikka voimme nähdä, mitä ohjelmoija tarkoittaa, tarvitsee kääntäjä hieman apua. Jotta voisimme käsitellä kantaluokan jäsentä nimellä, jota on käytetty uudelleen johdetussa luokassa, pitää kantaluokan jäsen tarkentaa luokkansa viittausalueoperaattorilla. Tässä on esimerkiksi turn_eyes_down():n oikea uudelleentoteutus:

```
void
Shy::
turn_eyes_down()
{
   // ...
   _mumble = "excuse me"; // ok

   // ok: tarkennettu kantaluokan ilmentymään
   Diffident::_mumble = -1;
}
```

Niille, joille kieli on uusi asia, on yleistä virhekäsitys, että kantaluokan ja johdetun luokan jäsenfunktiot muodostuvat ylikuormitetuista funktioista. Esimerkiksi:

```
class Diffident {
public:
   void mumble( int softness );
   // ...
};

class Shy : public Diffident {
public:
   // sananmukaisesti peittää Diffident::mumble:n näkyvyyden
   // ne eivät muodosta ylikuormitettua ilmentymäparia
   void mumble( string whatYaSay );
   void print( int soft, string words );
   // ...
};
```

Yritys käynnistää kantaluokan ilmentymä johdetussa luokassa johtaa kuitenkin käännösvirheeseen. Esimerkiksi:

```
Shy simon;

// ok: Shy::mumble( string )
simon.mumble( "pardon me" );

// virhe: odotetaan, että ensimmäinen argumentti on string-tyyppinen
// Diffident::mumble( int ) ei ole näkyvissä
simon.mumble( 2 );
```

Vaikka kantaluokan jäseniä voidaan käsitellä suoraan, ne säilyttävät sen kantaluokan viittausalueen, jossa ne on määritelty. Kaikkien niiden ylikuormitettujen funktioehdokkaitten nimien pitää esiintyä samalla viittausalueella. Ellei niin olisi asianlaita, johtaisivat seuraavat kaksi ilmentymää ei-virtuaalisesta jäsenfunktiosta turn_aside()

```
class Diffident {
public:
   void turn_aside();
   // ...
};

class Shy : public Diffident {
public:
   // sananmukaisesti peittää
   // Diffident::turn_aside():n näkyvyyden
   void turn_aside();

   // ...
};
```

uudelleenmäärittelyvirheeseen, koska molemmilla ilmentymillä on samanlainen tunniste. Ne eivät ole virheellisiä, koska molemmat sijaitsevat sen luokan viittausalueella, jossa ne on määritelty.

Mitä jos todella haluaisimme lisätä ilmentymien ylikuormitetun joukon sekä johdetuista että kantaluokan jäsenistä? Pitääkö meidän kirjoittaa pieniä välittömiä funktiopätkiä johdettuun luokkaan käynnistämään kantaluokan ilmentymää? Vaikka tämä toteuttaa päämäärämme

```
class Shy : public Diffident {
public:
   // ok: eräs tapa tehdä ylikuormitettu joukko
   //     kantaluokan ja johdetun luokan jäsenistä

   void mumble( string whatYaSay );
   void mumble( int softness ) {
            Diffident::mumble( softness ); }
   // ...
};
```

se on tarpeetonta C++-standardissa. Pääsemme samaan tulokseen *using-esittelyllä*, kuten seuraavassa:

```
class Shy : public Diffident {
public:
   // ok: C++-standardissa using-esittely
   //     luo ylikuormitetun joukon
   //     kantaluokan ja johdetun luokan jäsenistä

   void mumble( string whatYaSay );
   using Diffident::mumble;

   // ...
};
```

Itse asiassa using-esittely tuo kantaluokan jokaisen nimetyn jäsenen johdetun luokan viittausalueelle. Kantaluokan jäsen tuodaan nyt ylikuormitettujen ilmentymien joukkoon, jonka nimi liittyy johdetun luokan jäsenfunktion nimeen. (Jäsenfunktion using-esittelyyn ei voida määrittää parametriluetteloa, vain jäsenfunktion nimi. Tämä merkitsee, että jos funktio on ylikuormitettu kantaluokassa, lisätään kaikki ylikuormitetut ilmentymät johdetun luokkatyypin viittausalueelle. Emme voi lisätä vain yhtä kantaluokan jäsenten ylikuormitettua ilmentymää.)

Toinen yleinen C++-ohjelmoijien virhekäsitys on ulottuvuudesta, kuinka kauas he voivat käsitellä kantaluokan suojattuja jäseniä. Kun kirjoitamme

```
class Query {
public:
   const vector<location>* locations() const { return &_loc; }
   // ...
protected:
     vector<location> _loc;
     // ...
};
```

tarkoitamme, että Query:stä johdettu luokka voi käsitellä suoraan _loc-tietojäsentä, kun taas muun ohjelman pitää käyttää julkista käsittelyfunktiota. Se, mitä tämä merkitsee kuitenkin, on se, että johdetulla luokalla on pääsy *oman* kantaluokka-aliolionsa suojattuun _loc-tietojäseneen. Johdetulla luokalla ei ole pääsyä itsenäisen kantaluokkaolion suojattuihin jäseniin. Esimerkiksi:

```
bool
NameQuery::
compare( const Query *pquery )
{
   // ok: sen Query-aliolion suojattu jäsen
   int myMatches = _loc.size();

   // virhe: ei ole suoraa pääsyoikeutta
   // riippumattoman Query-olion suojattuun jäseneen
   int itsMatches = pquery->_loc.size();
```

```
    return myMatches == itsMatches;
}
```

NameQuery:llä on pääsy vain yhden Query-luokkaolion suojattuihin jäseniin: sen omaan Query-aliolioon. (Näitä suojattuja jäseniä käsitellään johdetussa luokassa implisiittisen this-osoittimen kautta [this-osoitin esiteltiin kohdassa 13.4].) Välitön ratkaisu käännösvirheeseen on kirjoittaa compare()-funktio uudelleen, jotta se käyttäisi hyväkseen julkista location()-jäsenfunktiota:

```
bool
NameQuery::
compare( const Query *pquery )
{
    // ok: sen Query-aliolion suojattu jäsen
    int myMatches = _loc.size();

    // ok: käytä julkista käsittelymetodia
    int itsMatches = pquery->locations()->size();

    return myMatches == itsMatches;
}
```

Todellinen ongelma on kuitenkin suunnitteluvirhe omalta osaltamme. Koska _loc on Query-kantaluokan jäsen, compare() kuuluu jäsenenä oikeastaan Query-luokkaan eikä johdettuun NameQuery-luokkaan. Usein jäsenen käsittelyongelmat johdetun luokan ja kantaluokan välillä ratkaistaan siirtämällä operaatio luokkaan, joka sisältää tavoittamattomissa olevan jäsenen, kuten tässä tapauksessa.

Tämän tyyppinen luokan jäsenen käsittelyrajoitus ei päde muille sen luokkaolioille. Esimerkiksi:

```
bool
NameQuery::
compare( const NameQuery *pname )
{
    int myMatches = _loc.size(); // ok
    int itsMatches = name->_loc.size(); // ok yhtä hyvin

    return myMatches == itsMatches;
}
```

Johdettu luokka voi käsitellä suoraan oman luokkansa muiden olioiden kantaluokan suojattuja jäseniä kuten myös luokkansa muiden olioiden suojattuja ja yksityisiä jäseniä.

Mietipä seuraavaa Query-kantaluokan osoittimen alustusta osoitteella, joka on johdetun NameQuery-olion osoite:

```
Query *pb = new NameQuery( "sprite" );
```

Jos käynnistämme virtuaalisen funktion, joka on määritelty Query-kantaluokkaan kuten

```
pb->eval(); // käynnistää NameQuery::eval():in
```

ei johdettua NameQuery-luokkailmentymää käynnistetä. Paitsi Query-kantaluokassa esitellylle virtuaalifunktiolle, joka korvautuu johdetussa NameQuery-luokassa, ei ole olemassa tapaa käsitellä NameQuery:n jäsentä suoraan pb:n kautta:

1. Jos Query ja NameQuery molemmat esittelevät samannimisen ei-virtuaalisen jäsenfunktion, Query-ilmentymä käynnistetään aina pb:n kautta.
2. Samalla tavalla, jos Query ja NameQuery molemmat esittelevät samannimisen tietojäsenen, käsitellään Query-ilmentymää aina pb:n kautta.
3. Jos NameQuery esittelee virtuaalifunktion, jota ei ole Query:ssä, kuten esimerkiksi suffix(), yritys käynnistää se pb:n kautta saa aikaan käännösvirheen:

   ```
   // virhe: suffix() ei ole Query:n jäsen
   pb->suffix();
   ```

4. Samalla tavalla, jos yritämme käsitellä NameQuery:n ei-virtuaalista jäsenfunktiota tai tietojäsentä pb:n kautta, on tuloksena käännösvirhe:

   ```
   // virhe: _name ei ole Query:n jäsen
   pb->_name;
   ```

   Käsiteltävän jäsenen tarkennus ei auta tässä tapauksessa:

   ```
   // virhe: Query:llä NameQuery-kantaluokkaa
   pb->NameQuery::name();
   ```

C++:ssa kantaluokan osoitin voi käsitellä vain tietojäseniä ja jäsenfunktioita mukaan lukien virtuaaliset jäsenfunktiot, jotka on esitelty (tai peritty) sen luokassa huolimatta todellisesta oliosta, jota se osoittaa. Jäsenfunktion esittely virtuaaliseksi vain viivyttää käynnistettävän ilmentymän ratkaisua, joka perustuu todelliseen luokkatyyppiin, jota pb osoittaa ohjelman suorituksen aikana.

Vaikka tämä voi näyttää joustamattomalla, tuo se kaksi merkittävää etua:

1. Virtuaalifunktion suorituksenaikainen käynnistys ei voi koskaan epäonnistua, koska todellisen luokkatyypin ilmentymää ei ole olemassa. Jos jotain tiettyä ilmentymää ei ole olemassa, ei ohjelmaa voi kääntää.
2. Virtuaalimekanismia voidaan optimoida. Virtuaalifunktion kutsu ei useinkaan ole raskaampaa kuin funktion käynnistys epäsuorasti osoittimen kautta (katso julkaisusta [LIPPMAN96a] aiheen koko käsittely). (Katsomme virtuaalifunktioita tarkemmin kohdassa 17.5.)

Query-kantaluokkaan on määritelty staattinen tietojäsen _text_file:

```
static vector<string> *_text_file;
```

Luoko johdettu luokka, NameQuery, toisen ilmentymän _text_file-tietojäsenestä, joka on yhtäläinen NameQuery-luokalle? Ei. Kaikki johdetut luokkaoliot viittaavat samaan, yhteen,

jaettuun staattiseen jäseneen. Huolimatta Query:stä johdettujen luokkien lukumäärästä, on olemassa yksi _text_file-ilmentymä. Jos haluamme, voimme käsitellä sitä johdetun luokkaolion kautta käyttäen jäsenen käsittelysyntaksia:

```
nameQueryObject._text_file; // ok
```

Lopuksi, jos johdettu luokka haluaa käsitellä kantaluokkansa yksityisiä jäseniä suoraan, pitää kantaluokan esitellä johdettu luokka eksplisiittisesti ystäväksi. Esimerkiksi:

```
class Query {
    friend class NameQuery;
public:
  // ...
};
```

NameQuery voi nyt käsitellä, ei vain oman kantaluokka-aliolionsa yksityisiä jäseniä, vaan myös kaikkien Query-olioiden yksityisiä ja suojattuja jäseninä.

Mitä jos johtaisimme StringQuery-luokan NameQuery:stä? StringQuery tukee lyhennettyä AndQuery-ilmaisua niin, että kun kirjoitetaan

```
beautiful && fiery && bird
```

voi käyttäjä yksinkertaisesti kirjoittaa

```
"beautiful fiery bird"
```

Periikö StringQuery NameQuery:n ja Query:n välisen ystävyyden? Ei. Ystävyyttä ei peritä. Johdetusta luokasta ei tule sen luokan ystävää, joka myönsi ystävyyden jollekin kantaluokistaan. Jos johdettu luokka vaatii yhden tai useamman samoja ystävyyksiä, täytyy vastaavien luokkien myöntää ne eksplisiittisesti. Esimerkiksi StringQuery:llä ei ole erityispääsyoikeutta Query:yn. Jos se tarvitsee erikoispääsyn, pitää Query:n myöntää se eksplisiittisesti.

Harjoitus 17.6

Olkoot seuraavat kantaluokan ja johdetun luokan määrittelyt:

```
class Base {
public:
    foo( int );
    // ...
protected:
    int _bar;
    double _foo_bar;
};

class Derived : public Base {
public:
    foo( string );
    bool bar( Base *pb );
    void foobar();
```

```
    // ...
protected:
    string _bar;
};
```

Yksilöi, mitä on pielessä jokaisessa seuraavassa koodikatkelmassa, ja kuinka se voitaisiin korjata:

```
(a) Derived d; d.foo( 1024 );
(b) void Derived::foobar() { _bar = 1024; }
(c) bool Derived::bar( Base *pb )
        { return _foo_bar == pb->_foo_bar; }
```

17.4 Kantaluokan ja johdetun luokan muodostus

Muista, että johdettu luokka muodostuu yhdestä tai useammasta kantaluokka-alioliosta ja johdetun luokan osasta. Esimerkiksi NameQuery muodostuu Query-alioliosta ja string-jäsenluokkaoliosta. Koska tarkoituksena on kuvata johdetun luokan muodostajan käyttäytymistä, esittelemme myös sisäisen tietojäsenen:

```
class NameQuery : public Query {
public:
  // ...
protected:
    bool   _present;
    string _name;
};
```

Jos _present on asetettu arvoon epätosi, se ilmaisee, että _name:a ei löydy tekstistä.

Miettikäämme ensin tilannetta, jolloin emme määrittelisi NameQuery-luokan muodostajaa. Siinä tapauksessa, kun määrittelemme NameQuery-olion

```
NameQuery nq;
```

käynnistetään ensin Query-luokan oletusmuodostaja, sitten string-luokan oletusmuodostaja (liittyy jäsenluokkaolioon _name). _present jää alustamatta, mikä on potentiaalinen ohjelmavirheen lähde.

Jotta alustaisimme _present-jäsenen, voimme määritellä NameQuery:n oletusmuodostajan kuten seuraavassa:

```
inline NameQuery::NameQuery(){ _present = false; }
```

Nyt nq-määrittely käynnistää kolme muodostajaa: Query-kantaluokan oletusmuodostajan, string-oletusmuodostajan _name-tietojäsenen alustukseen ja NameQuery:n oletusmuodostajan.

Mitä, jos haluamme välittää argumentin Query-kantaluokan muodostajalle? Kuinka voisimme sen tehdä? Voimme vastata tähän yhdenmukaisuussyiden kautta:

Voidaksemme välittää yhden tai useamman argumentin jäsenluokkaolion muodostajalle, teemme sen jäsenen alustusluettelon kautta (voimme alustaa myös luokattomia tietojäseniä samalla tavalla — katso aiheen käsittely kohdasta 14.5). Esimerkiksi:

```
inline NameQuery::
NameQuery( const string &name )
        : _name( name ), _present( false )
{}
```

Voidaksemme välittää yhden tai useamman argumentin kantaluokan muodostajalle, teemme sen myös jäsenen alustusluettelolla. Seuraavassa esimerkissä välitämme string-muodostajalle name-argumentin ja Query-kantaluokan muodostajalle olion, jota ploc osoittaa:

```
inline
NameQuery::
NameQuery( const string &name,
        vector<location> *ploc )
         : _name( name ), Query( *ploc ), _present( true )
       {}
```

Vaikka Query on sijoitettu toiseksi jäsenen alustusluettelossa, se käynnistetään aina ennen _name:en liittyvää string-muodostajaa. Muodostajien käynnistysjärjestys on aina seuraava:

1. Kantaluokan muodostaja. Jos kantaluokkia on useampi kuin yksi, muodostajat käynnistetään siinä järjestyksessä, jossa kantaluokat esiintyvät johdettujen luokkien luettelossa eikä siinä järjestyksessä, jossa ne on lueteltu jäsenen alustusluettelossa. (Katsomme moniperiytymistä luvussa 18.)
2. Jäsenluokkaolion muodostaja. Jos on useampi kuin yksi jäsenluokkaolio, muodostajat käynnistetään siinä järjestyksessä, jossa oliot on esitelty luokassa eikä siinä järjestyksessä, jossa ne on lueteltu jäsenen alustusluettelossa (katso kohdasta 14.5 tarkempi käsittely).
3. Johdetun luokan muodostaja.

Yleisenä sääntönä on, että johdetun luokan muodostajan ei tulisi koskaan sijoittaa arvoa kantaluokan tietojäseneen suoraan, vaan välittää arvo kyseiselle kantaluokan muodostajalle. Muussa tapauksessa noiden kahden luokan toteutuksesta tulee *tiiviisti yhteenkytkettyjä* (*tightly coupled*) ja voi olla yhä vaikeampaa muokata tai laajentaa kantaluokan toteutusta oikein. (Kantaluokan suunnittelijoina meillä on vastuu tehdä sopivasti kantaluokan muodostajia.)

Tämän kohdan loppuun saakka käymme vuorollaan läpi Query-kantaluokan ja neljän johdetun luokan muodostajien suunnittelun. Kun se on tehty, mietimme lyhyesti vaihtoehtoista Query-hierarkiaa, jossa hierarkioita on syvemmälti kuin kaksi. Päätämme tämän kohdan katsauksella luokan tuhoajiin.

17.4.1 Kantaluokan muodostaja

Query-luokassamme on esitelty kaksi ei-staattista tietojäsentä: _solution ja _loc:

```
class Query {
public:
   // ...
protected:
      set<short> *_solution;
      vector<location> _loc;
   // ...
};
```

Query-luokkamme oletusmuodostajan tarvitsee alustaa vain _solution eksplisiittisesti. Vektorin oletusmuodostaja käynnistetään automaattisesti alustamaan _loc. Tässä on sen toteutus:

```
inline Query::Query(): _solution( 0 ) {}
```

Myös toinen Query-luokan muodostaja pitää määritellä. Tämä saa viittauksen paikkavektoriin kuten seuraavassa:

```
inline
Query::
Query( const vector< location > &loc )
      : _solution( 0 ), _loc( loc )
{}
```

Tämä toinen Query:n muodostaja käynnistetään vain NameQuery-muodostajassa, kun NameQuery-edustaa tekstissä olevaa sanaa. Siinä tapauksessa välitetään esilaskettu paikkavektori sanan kanssa. Kolme muuta johdettua luokkatyyppiä laskevat paikkavektorinsa omissa eval()-jäsenfunktiossaan. (Näemme esimerkin tästä seuraavassa alikohdassa. eval()-jäsenfunktioiden toteutukset on esitetty kohdassa 17.5, kun käsittelemme virtuaalifunktioita.)

Kysymys kuuluu nyt, millä käsittelytasolla muodostajat tulisi esitellä? Emme halua esitellä niitä julkisina, koska Query-luokkaolion on tarkoitus olla ohjelmassamme jonkin sen johdetun alityypin luokkaoliossa alioliona. Ilmaisemme tämän esittelemällä muodostajan suojatuksi julkisen sijasta:

```
class Query {
public:
   // ...
protected:
   Query();
   // ...
};
```

Toinen Query-muodostaja käynnistetään jopa rajoittavammassa tilanteessa: sen ei tulisi vain muodostaa Query-alioliota, vaan sen tulisi muodostaa vain NameQuery-olion Query-aliolio. Voimme varmistua tästä esittelemällä tämän toisen muodostajan yksityiseksi ja esittelemällä NameQuery-luokan Query-luokan ystäväksi. (Kuten kerroimme edellisessä kohdassa, johdettu luokka voi käsitellä kantaluokkansa vain julkisia ja suojattuja jäseniä. Kaikki yritykset käyn-

nistää OrQuery-, AndQuery- ja NotQuery-luokassa toinen muodostaja johtavat nyt käännösvirheeseen.)

```
class Query {
      friend class NameQuery;
public:
   // ...
protected:
   Query();
   // ...
private:
   explicit Query( const vector<location>& );
};
```

(Joku voi väittää tätä toisen muodostajan ilmentymää vastaan ja ehdottaa, että on parempi täyttää _loc NameQuery:n eval()-funktiolla. Tämä suunnitelma, jossa on kaksi muodostajaa, sopii kuitenkin paremmin tarkoituksiimme kuvata kantaluokan muodostajan käyttöä.)

17.4.2 Johdetun luokan muodostaja

NameQuery-luokkamme määrittelee myös kaksi muodostajaa. Niistä on tehty julkisia, koska oletetaan, että todelliset NameQuery-oliot määritellään sovelluksessamme.

```
class NameQuery : public Query {
public:
   explicit NameQuery( const string& );
   NameQuery( const string&, vector<location>* );
   // ...
protected:
   // ...
};
```

Yksiparametrinen muodostaja saa string-parametrin. Tämä välitetään string-muodostajalle, joka käynnistetään alustamaan string-tietojäsen _name. Query-kantaluokan oletusmuodostaja käynnistetään implisiittisesti:

```
inline
NameQuery::
NameQuery( const string &name )
     // Query::Query() käynnistetty implisiittisesti
     : _name( name )
{}
```

Kaksiparametrinen muodostaja saa myös string-parametrin. Se saa myös toisen parametrin, jonka tyyppi on osoitin paikkavektoriin. Tämä välitetään yksityiselle Query-kantaluokan muodostajalle. (Huomaa, että emme enää pidä _present:iä NameQuery:n tietojäsenenä):

```
inline
NameQuery::
NameQuery( const string &name, vector<location> *ploc )
```

```
        : _name( name ), Query( *ploc )
{}
```

Tässä on esimerkki, kuinka niitä voitaisiin käyttää:

```
string title( "Alice" );
NameQuery *pname;

// katso, esiintyykö "Alice" sanojen tekstikartassa
// jos esiintyy, hae sitä vastaava paikkavektori

if ( vector<location> *ploc = retrieve_location( title ))
    pname = new NameQuery( title, ploc );
else pname = new NameQuery( title );
```

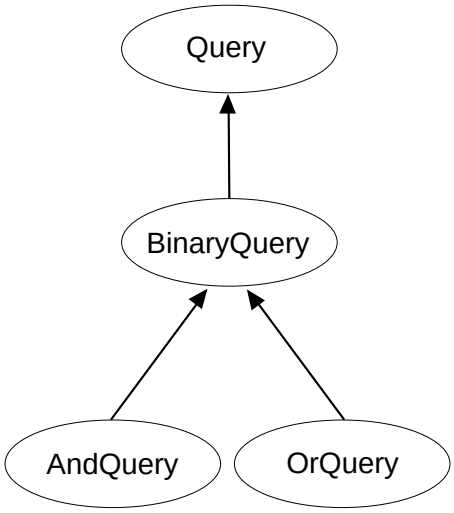
NotQuery-, OrQuery- ja AndQuery-luokissa määritellään jokaisessa yksi muodostaja kuten seuraavassa, jossa Query-kantaluokan oletusmuodostaja on käynnistetty implisiittisesti:

```
inline NotQuery::
NotQuery( Query *op = 0 ) : _op( op ) {}

inline OrQuery::
OrQuery( Query *lop = 0, Query *rop = 0 )
    : _lop( lop ), _rop( rop )
{}

inline AndQuery::
AndQuery( Query *lop = 0, Query *rop = 0 )
    : _lop( lop ), _rop( rop )
{}
```

Kohdassa 17.7 muodostamme yksilölliset johdetut luokkaoliot edustamaan jokaista käyttäjän kyselyä.

17.4.3 Vaihtoehtoinen luokkahierarkia

Vaikka Query-luokkahierarkiamme on riittävä suunnitelma, se ei ole ainoa mahdollinen. Koska esimerkiksi AndQuery- ja OrQuery-luokat molemmat tukevat binäärioperaatiota, on näiden luokkien välillä hieman asioita tuplasti. Voimme laittaa näiden kahden luokan yhteiset tietojäsenet ja jäsenfunktiot abstraktiin BinaryQuery-kantaluokkaan. Query-hierarkiamme uusi alipuu on kuvassa 17.2.



Kuva 17.2 Vaihtoehtoinen luokkahierarkia

BinaryQuery-luokka on myös abstrakti kantaluokka — luokan todelliset oliot eivät esiinny sovelluksessamme. eval()-funktiolle ei ole mielekästä toteutusta, joten päätämme olla tekemättä määrittelyä Query-kantaluokassamme esitellystä virtuaalisesta ilmentymästä. BinaryQuery-luokan puhdas eval()-virtuaalifunktio on myös aktiivinen. (Katsomme puhtaita virtuaalifunktioita tarkemmin kohdassa 17.5.)

Kaksi käsittelyjäsentä, lop() ja rop(), jotka ovat yhteisiä molemmille johdetuille luokille, nostetaan BinaryQuery-luokkaan. Ne on määritelty ei-staattisiksi, välittömiksi funktioiksi. Samalla tavalla kaksi tietojäsentä, _lop ja _rop, jotka molemmat on esitelty johdetuissa luokissa, nostetaan BinaryQuery-luokkaan. Ne esitellään suojattuina, ei-staattisina tietojäseninä. Näiden kahden johdetun luokan julkiset muodostajat yhdistetään yhteen suojattuun BinaryQuery-muodostajaan:

```
class BinaryQuery : public Query {
public:
   const Query *lop() { return _lop; }
   const Query *rop() { return _rop; }

protected:
   BinaryQuery( Query *lop, Query *rop )
            : _lop( lop ), _rop( rop )
   {}

   Query *_lop;
   Query *_rop;
};
```

Näyttää aivan siltä, että näihin kahteen johdettuun luokkaan pitää nyt tehdä vain sopiva eval()-ilmentymä:

```
// hups: nämä luokkamäärittelyt ovat virheellisiä

class OrQuery : public BinaryQuery {
public:
    virtual void eval();
};

class AndQuery : public BinaryQuery {
public:
    virtual void eval();
};
```

Määrittelyistämme huolimatta ne ovat kuitenkin vielä epätäydellisiä. Yllätys on, että jos käännämme nämä kaksi luokkamäärittelyä, se tapahtuu ilman virheitä. Jos yritämme määritellä varsinaisen luokkaolion kuten tässä

```
// virhe: AndQuery-luokan muodostaja puuttuu
AndQuery proust( new NameQuery( "marcel" ),
            new NameQuery( "proust " ));
```

johtaa proust-määrittely virheeseen: meille kerrotaan, että AndQuery-luokalta puuttuu muodostajan määrittely, joka tukisi kahden argumentin välitystä.

Olimme olettaneet, että sekä AndQuery että OrQuery perivät BinaryQuery-muodostajan samalla tavalla kuin ne molemmat perivät lop()- ja rop()-jäsenkäsittelyfunktiot. Niin ne eivät tee kuitenkaan. Johdettu luokka ei peri kantaluokkansa muodostajia. (Syy tähän on se, että tämä voisi liian helposti johtaa johdetun luokan alustamattomien jäsenten esittelyvirheisiin. Kuvittele esimerkiksi, että seuraavassa lisäämme yhden tai toisen luokattoman tietojäsenen AndQuery:yn. Peritty kantaluokan muodostaja ei enää ole riittävä johdetun AndQuery-luokan alustamiseen. Ohjelmoija, joka lisää uuden jäsenen, ei ehkä kuitenkaan havaitse tätä. Virhe kuitenkin paljastaa itsensä, ei AndQuery-olion muodostamisessa, vaan jatkossa jossakin olion käyttötilanteessa. Käytännössä tällaiset virheet osoittautuvat vaikeiksi jäljittää. Kantaluokan new- ja delete-operaattorien ylikuormitetut ilmentymät periytyvät ja johtavat joskus juuri tämänkaltaisiin ongelmiin.)

Jokaisen johdetun luokan pitää tehdä omat muodostajansa. AndQuery- ja OrQuery-luokkiemme kohdalla muodostajat eivät toimi muussa tarkoituksessa kuin muodostavat rajapinnan, jolla välittää kaksi operandia BinaryQuery-muodostajalle. Tässä on korjattu toteutuksemme:

```
// ok: nämä luokkamäärittelyt ovat oikein

class OrQuery : public BinaryQuery {
public:
    OrQuery( Query *lop, Query *rop )
        : BinaryQuery( lop, rop ) {}
```

```
    virtual void eval();
};

class AndQuery : public BinaryQuery {
public:
    AndQuery( Query *lop, Query *rop )
        : BinaryQuery( lop, rop ) {}

    virtual void eval();
};
```

Jos katsomme kuvaa 17.2 jälleen, näemme, että BinaryQuery on AndQuery:n ja OrQuery:n lähin kantaluokka. Query on BinaryQuery:n lähin kantaluokka. Query ei ole AndQuery- ja OrQuery-luokkien lähin kantaluokka.

Johdetun luokan muodostaja voi käynnistää sallitusti vain lähimmän kantaluokkansa muodostajan (virtuaaliperiytyminen tarjoaa poikkeuksen tähän sääntöön, kuten myös muihinkin sääntöihin; katso kohtaa 18.5). On esimerkiksi virhe, jos AndQuery käynnistää Query:n muodostajan jäsenen alustusluettelollaan.

AndQuery- tai OrQuery-luokkaolioiden määrittelyt johtavat nyt seuraavan kolmen muodostajan käynnistämiseen: ei-lähimmän Query-kantaluokan muodostaja, lähimmän BinaryQuery-kantaluokan muodostaja ja johdetun AndQuery- tai OrQuery-luokan muodostaja. (Kantaluokan muodostajien käynnistysjärjestys kuvastaa johdetun luokan periytymishierarkian syvyys ensin -käyntijärjestystä.) Lisätyn BinaryQuery-luokkajohdannaisen vaikutus suorituskykyyn on mitätön, koska olemme määritelleet sen välittömäksi.

Koska muokattu hierarkia säilyttää alkuperäisen suunnitelman julkisen rajapinnan, ei muutos jo hierarkiaa hyväksikäyttävään koodiin ole hyökkäävä. Kuitenkin, vaikka lähdetason koodia ei tarvitse muokata, se pitää kääntää uudelleen käyttäen luokkahierarkiamme uutta määrittelyä. Tämän koko järjestelmän uudelleenkäännösvaatimus saattaa kuitenkin pelottaa joitakin käyttäjiä siirtymästä uuteen suunnitelmaan.

17.4.4 Laiskaa virheen selvitystä

Ohjelmoijat, joille C++ on uutta, saattavat joskus yllättyä, että AndQuery- ja OrQuery-luokkien kelpaamattomat määrittelyt (molemmista puuttui tarpeellisen muodostajan esittely) kääntyvät virheittä. Ellemme olisi yrittäneet määritellä todellista AndQuery-oliota, olisimme saattaneet levittää (nöyryyttävästi) eli antaa käyttäjien käyttöön muokatun luokkahierarkian havaitsematta virhettämme. Mitä oikein tapahtuu? Mietipä seuraavaa:

1. Jos virheestä ilmoitetaan esittelykohdassa, silloin emme voi jatkaa sovelluksemme käännöstä, ennen kuin virhe on korjattu. Jos kuitenkin virheen aiheuttava esittely on osa kirjastoa, jonka lähdekoodiin meillä ei ole pääsyä, saattaa ristiriidan ratkaisu olla todella suuri työ. Lisäksi voi olla niin, että meillä ei ehkä koskaan ole syytä laukaista virhettä sovelluksessamme niin, että vaikka esittely edustaa potentiaalista virhettä, sitä ei koskaan realisoida koodissamme.

2. Toisaalta, jos siitä ei ilmoiteta ennen kuin käyttötilanteessa, silloin koodimme on mahdollisesti täynnä laukaisemattomia kielivirheitä, joita varomaton ohjelmoija saattaa laukaista milloin tahansa. Tämän strategian mukaan koodimme kääntäminen onnistuneesti ei takaa, että se on vapaa kielivirheistä, vaan että ohjelmamme ei käytä hyväkseen kielen sääntörikkomusten koodia.

Virheen generointi käyttökohdassa on *laiskan arvioinnin* muoto, joka on yleinen suunnittelustrategia parantaa ohjelman suorituskykyä. Sitä käytetään usein potentiaalisesti raskaiden resurssien varaamisen tai alustamisen yhteydessä, kunnes niitä todella tarvitaan. Ellei niitä koskaan tarvita, säästämme sen tarpeettoman järjestelyn aiheuttaman kuorman. Jos resursseja tarvitaan, mutta ei kaikkea kerralla, silloin “kuoletamme” ohjelman järjestelykuormaa.

C++:ssa potentiaaliset “yhdistelmävirheet”, jotka liittyvät ylikuormitettuihin funktioihin, malleihin ja luokkaperiytymiseen yleensä, ilmoitetaan virheinä pikemminkin käyttökohdissa kuin esittelykohdissa. Se, oletko vakuuttunut siitä, että tämä on oikea strategia (käytännössä uskomme, että se on; jokaisen potentiaalisen virheen ratkaisu, kun useita komponentteja yhdistellään, ei yksinkertaisesti ole tuottavaa), on se strategia paikallaan. Tämä merkitsee, että omaa koodiamme pitää testata jatkuvasti löytääksemme ja ratkaistaksemme siinä piilevät virheet. Kun yhdistellään kaksi tai useampi suuri komponentti, ovat piilevät virheet marginaalisesti hyväksyttäviä. Yhdessä komponentissa kuten Query-luokkahierarkiassa piilevät virheet eivät yleensä ole hyväksyttäviä missään määrin.

17.4.5 Tuhoajat

Kun johdetun luokkaolion elinkaari päättyy, käynnistetään automaattisesti johdetun ja kantaluokan tuhoajat, jos ne on määritelty kuten myös jokaisen jäsenluokkaolion tuhoajat. Esimerkiksi seuraavassa NameQuery-luokkaoliossa

```
NameQuery nq( "hyperion" );
```

tuhoajajärjestys on seuraava: (1) NameQuery-luokan tuhoaja, (2) string-tuhoaja tietojäsenelle `_name` ja (3) Query-kantaluokan tuhoaja. Yleisemmin: johdettujen luokkaolioiden tuhoajien käynnistysjärjestys on päinvastainen kuin niiden muodostajien käynnistysjärjestys.

Tässä on Query-kantaluokkamme ja johdettujen luokkien tuhoajat (nämä tulisi kaikki esitellä vastaavien luokkien julkisiksi jäseniksi):

```
inline Query::
~Query(){ delete _solution; }

inline NotQuery::
~NotQuery(){ delete _op; }

inline OrQuery::
~OrQuery(){ delete _lop; delete _rop; }

inline AndQuery::
~AndQuery(){ delete _lop; delete _rop;  }
```

Tässä on kaksi asiaa, jotka tulisi huomioida: (1) emme tee eksplisiittistä NameQuery-tuhoajaa. Miksi? Koska ei ole ohjelmatasoista siivousta, joka pitää tehdä. Sekä Query-kantaluokan tuhoaja että _name:n string-luokan tuhoaja käynnistetään automaattisesti. (2) Johdettujen luokkien tuhoajissa käytetään delete-lauseketta Query*-osoittimeen. Mutta se ei ole se Query-tuhoaja, jonka halusimme käynnistää. Sen sijaan meidän pitää käynnistää osoittimen osoittaman todellisen luokkatyypin olion tuhoaja. Tämän toteuttamiseksi Query-kantaluokkamme pitää esitellä tuhoaja virtuaaliseksi. Katsomme virtuaalituhoajia ja virtuaalifunktioita yleisesti seuraavassa kohdassa.

On olemassa vielä eräs asia, joka tulisi mainita. Toteutuksemme implisiittinen olettamus on, että operandit, joihin NotQuery-, OrQuery- ja AndQuery-luokkaolioissa viitataan, on varattu keosta. Tästä syystä käytämme delete-operaattoria jokaiseen operandiin vastaavissa tuhoajissa. Tämä ei kuitenkaan ole pakko-olettamus kielitasolla. Kieli ei erota toisistaan osoitteita keosta tai muualta. Eräässä mielessä toteutuksemme on siten turvaton.

Kuten tulemme näkemään kohdassa 17.7, olemme kapseloineet Query-hierarkian varaamisen ja muodostamisen UserQuery-manager-luokkaan. Tämä antaa meille riittävän luottamuksen — ainakin tässä *C++-kirjassa* oletamme, että luottamustamme vastaan ei rikota. Yleiskäyttöiselle kirjastiolle vaaditaan kuitenkin lisätakuita. Ohjelmatasoinen pakotusstrategia on ylikuormittaa hierarkialuokkien new- ja delete-operaattorit. Eräs mahdollinen ohjelmatasoinen strategia on seuraava: new-operaattorit merkitsevät, että oliot on varattu keosta. Sitten ne varaavat olion new-lauseketta käyttäen. delete-operaattorit tarkistavat new-operaattorien olemassaolon. Jos ne ovat olemassa, käytetään delete-lauseketta operandina.

Harjoitus 17.7

Yksilöi kantaluokan ja johdettujen luokkien muodostajat ja tuhoajat luokkahierarkialle, joka valittiin kohdan 17.1 lopussa harjoituksessa 17.2.

Harjoitus 17.8

Toteuta uudelleen OrQuery-luokka niin, että johdat sen abstraktista UnaryQuery-luokasta.

Harjoitus 17.9

Mitä on pielessä seuraavassa luokkamäärittelyssä?

```
class Object {
public:
    virtual ~Object();
    virtual string isA();
protected:
    string _isA;
private:
    Object( string s ) : _isA( s ){}
};
```

Harjoitus 17.10

Olkoon seuraava kantaluokan määrittely:

```
class ConcreteBase {
public:
    explicit ConcreteBase( int );
    virtual ostream& print( ostream& );
    virtual ~Base();

    static int object_count();
protected:
    int _id;
    static int _object_count;
};
```

Mitä on pielessä seuraavissa?

```
(a) class C1 : public ConcreteBase {
  public:
      C1( int val )
       : _id( _object_count++ ){}
      // ...
  };

(b) class C2 : public C1 {
  public:
      C2( int val )
    : ConcreteBase( val ), C1( val ){}
      // ...
  };

(c) class C3 : public C2 {
  public:
      C3( int val )
    : C2( val ), _object_count( val ){}
      // ...
  };

(d) class C4 : public ConcreteBase {
  public:
        C4( int val )
         : ConcreteBase( _id+val ){}
        // ...
  };
```

Harjoitus 17.11

Alkuperäisessä C++:n määrittelyssä jäsenen alustusluettelon alustusjärjestys pääteltiin muodostajien käynnistysjärjestyksestä. Tämä muutettiin nykyiseksi kielen säännöksi vuonna 1986. Mitä luulet, miksi alkuperäistä kielen sääntöä muutettiin?

17.5 Kantaluokan ja johdetun luokan virtuaaliset funktiot

Oletusarvo on, että luokan jäsenfunktiot eivät ole virtuaalisia. Kun jäsenfunktio ei ole virtuaalinen, käynnistettävä funktio on se, joka on määritelty luokkaolion staattiseksi tyypiksi (eli osoitin tai viittaus), jonka kautta se käynnistetään. Esimerkiksi:

```
void Query::display( Query *pb )
{
   set<short> *ps = pb->solutions();
   // ...
   display();
}
```

pb:n staattinen tyyppi on Query*. Käynnistettävä ei-virtuaalinen funktio solutions() on Query:n jäsenfunktio. Ei-virtuaalinen display()-funktio käynnistetään implisiittisen this-osoittimen kautta. this-osoittimen staattinen tyyppi on myös Query*. Käynnistettävä funktio on myös Query:n jäsenfunktio.

Jos haluamme esitellä funktion virtuaaliseksi, niin määritämme yksinkertaisesti avainsanan virtual:

```
class Query {
public:
   virtual ostream& print( ostream& = cout ) const;
   // ...
};
```

Kun jäsenfunktio on virtuaalinen, käynnistettävä funktio on se, joka on määritelty luokkaolion dynaamisessa tyypissä (eli osoitin tai viittaus), jonka kautta se käynnistetään. Sattumalta kuitenkin luokkaolion staattinen ja dynaaminen tyyppi ovat sama asia. Virtuaalifunktiomekanismi toimii odotustemme mukaisesti vain, kun niitä käytetään osoittimien tai viittausten kanssa.

Monimuotoisuus on nimittäin mahdollista vain, kun johdettua luokka-alityyppiä osoitetaan epäsuorasti kantaluokan joko viittauksen tai osoittimen kautta. Kantaluokkaolion käyttö ei säilytä johdetun luokan tyyppi-identiteettiä. Katsotaanpa esimerkiksi seuraavaa koodikatkelmaa:

```
NameQuery nq( "lilacs" );

// ok: mutta nq 'viipaloitu' Query:n alioliokSi
Query qobject = nq;
```

qobject:in alustaminen nq:lla on täysin sallittua: qobject on nyt yhtä kuin Query-kantaluokan nq-aliolio. qobject ei kuitenkaan ole NameQuery-olio. nq:n NameQuery-osuus on kirjaimellisesti

"*viipaloitu pois*" ennen qobject:in alustamista. NameQuery-luokan osuus ei suoraan mahdu muistialueeseen, joka on varattu Query-oliolle. Eräs C++:n oliosuuntautuneen ohjelmoinnin ironia on, että meidän pitää käyttää osoittimia ja viittauksia sen tukemiseen, mutta ei olioita. Esimerkiksi seuraavassa koodikatkelmassa

```
void print( Query object,
        const Query *pointer,
        const Query &reference )
{
   // ei voida päätellä ennen kuin suorituksen aikana
   // todellista käynnistettävää print()-ilmentymää
   pointer->print();
   reference.print();

   // käynnistää aina Query::print():in
   object.print();
}

int main()
{
   NameQuery firebird( "firebird" );
   print( firebird, &firebird, firebird );
}
```

kaksi käynnistystä pointer:in ja reference:n kautta ratkaistaan niiden dynaamisista tyypeistä. Tässä esimerkissä ne molemmat käynnistävät NameQuery::print():in. Käynnistys object:in kautta käynnistää aina Query::print():in. (Näemme ohjelmakoodiesimerkin viipaloitumisefektistä kohdassa 18.6.2.)

Seuraavissa alikohdissa kuvaamme virtuaalifunktioiden määrittelyä ja käyttöä käymällä läpi useiden ilmentymien toteutuksen. Jokainen virtuaalinen jäsenfunktio kuvastaa eri kantilta oliokeskeistä suunnittelua.

17.5.1 Virtuaalinen syöttö ja tulostus

Ensimmäinen virtuaalinen operaatio, jonka haluaisimme tehdä, on kyselyn tulostaminen joko vakiovirtaan tai tiedostoon:

```
ostream& print( ostream &os = cout ) const;
```

print() pitää esitellä virtuaaliseksi, koska jokainen toteutus on tyyppiriippuvainen ja se pitää pystyä käynnistämään Query*-osoittimen kautta. Esimerkiksi AndQuery:n print() voisi näyttää seuraavalta:

```
ostream&
AndQuery::print( ostream &os ) const
{
   _lop->print( os );
   os << " && ";
   _rop->print( os );
```

```
}
```

Funktio print() pitää esitellä niin, että se on abstraktin Query-kantaluokan virtuaalinen funktio. Muussa tapauksessa emme voi käynnistää print()-funktiota Query*-operandien AndQuery-, OrQuery- ja NotQuery-luokkien tietojäsenten kautta. Mutta print()-funktiolle ei ole mielekästä toteutusta Query-kantaluokassa. Nyt määrittelemme sen yksikertaisesti tyhjäksi funktioksi. (Myöhemmin määrittelemme sen uudelleen puhtaaksi virtuaaliseksi funktioksi.)

```
class Query {
public:
    virtual ostream& print( ostream &os=cout ) const {}
    // ...
};
```

Kun kantaluokka esittelee ensimmäisen kerran virtuaalisen funktion, pitää sen määrittää virtual-avainsana luokan esittelyyn. Jos määrittely on sijoitettu luokan ulkopuolelle, ei virtual-avainsanaa saa määrittää uudelleen. Esimerkiksi seuraava print()-määrittely johtaa käännösvirheeseen:

```
// virhe: avainsana virtual voi esiintyä vain
//        luokkamäärittelyssä
virtual ostream& Query::print( ostream& ) const { ... }
```

Oikeaan määrittelyyn ei pidä ottaa mukaan virtual-avainsanaa.

Luokka, joka esittelee virtuaalifunktion, pitää joko määritellä tai esitellä se puhtaaksi virtuaalifunktioksi (jälleen: tässä vaiheessa määrittelemme sen tyhjäksi funktioksi). Johdettu luokka voi joko tehdä oman ilmentymän, josta tulee sitten tuon funktion aktiivinen ilmentymä tai se voi periä kantaluokan aktiivisen ilmentymän. Jos johdettuun luokkaan on määritelty ilmentymä, sanotaan sen *kumoavan* kantaluokan ilmentymän.

Ennen kuin käymme läpi neljän johdetun luokan toteutukset, pitää ensin miettiä sulkujen mukanaoloa kyselyssä. Esimerkiksi kyselyssä

```
fiery && bird || shyly
```

käyttäjä etsii mitä tahansa esiintymää sanaparille

```
fiery bird
```

tai yhtä adverbia

```
shyly
```

Toisaalta kysely

```
fiery && ( bird || hair )
```

etsii mitä tahansa esiintymää joko sanaparille

```
fiery bird
```

tai

```
fiery hair
```

Elleivät meidän toteutuksemme print()-ilmentymästä toista alkuperäisiä sulkuja, on niillä pieni arvo käyttäjälle. Jotta voisimme pitää kirjaa tarpeellisista vasemmanpuoleisista ja oikeanpuoleisista suluista generointia varten, laitamme ei-staattisen tietojäsenparin abstraktiin Query-kantaluokkaamme yhdessä niitä tukevien käsittelyfunktioiden kanssa (tämänkaltainen jäsenten lisääminen jälkeenpäin on normaalia luokkahierarkian kehittymistä):

```
class Query {
public:
   // ...

   // aseta _lparen ja _rparen
   void lparen( short lp ) { _lparen = lp; }
   void rparen( short rp ) { _rparen = rp; }

   // hae _lparen- ja _rparen-arvot
   short lparen() { return _lparen; }
   short rparen() { return _rparen; }

   // tulosta vasen ja oikea sulku
   void print_lparen( short cnt, ostream& os ) const;
   void print_rparen( short cnt, ostream& os ) const;

protected:

   // pidä yllä vasemmanpuoleisten ja oikeanpuoleisten sulkujen lukumäärää
   short _lparen;
   short _rparen;
   // ...
};
```

_lparen edustaa vasemmanpuoleisten sulkujen lukumäärää, joka tietyn olion tulisi tulostaa; _rparen edustaa oikeanpuoleisten sulkujen lukumäärää. (Kohdassa 17.7 näemme, kuinka nämä luvut lasketaan ja kuinka näihin kahteen jäseneen sijoitetaan.) Tässä on esimerkki paljon “sulutetun” kyselyn käsittelystä:

```
==> ( untamed || ( fiery || ( shyly ) ) )
evaluate word: untamed
_lparen: 1
_rparen: 0

evaluate Or
_lparen: 0
_rparen: 0

evaluate word: fiery
_lparen: 1
_rparen: 0

evaluate Or
```

```
_lparen: 0
_rparen: 0

evaluate word: shyly
_lparen: 1
_rparen: 0

evaluate right parens:
_rparen: 3

( untamed ( 1 ) lines match
( fiery ( 1 ) lines match
( shyly ( 1 ) lines match
( fiery || ( shyly ( 2 ) lines match[3]
( untamed || ( fiery || ( shyly ))) ( 3 ) lines match

Requested query: ( untamed || ( fiery || ( shyly )))
( 3 ) like a fiery bird in flight. A beautiful fiery bird, he tells her,
( 4 ) magical but untamed. "Daddy, shush, there is no such thing,"
( 6 ) Shyly, she asks, "I mean, Daddy, is there?"
```

Tässä on NameQuery-toteutuksemme:

```
ostream&
NameQuery::
print( ostream &os ) const
{
   if ( _lparen )
       print_lparen( _lparen, os );

   os << _name;

   if ( _rparen )
       print_rparen( _rparen, os );

   return os;
}
```

Tässä on sen esittely:

```
class NameQuery : public Query {
public:
     virtual ostream& print( ostream &os ) const;
     // ...
};
```

Jotta johdetun luokan virtuaalifunktion ilmentymä voisi kumota kantaluokkansa aktiivisen ilmentymän, pitää sen prototyypin vastata täsmälleen kantaluokassa olevaa ilmentymää. Jos

3. Hups. Oikeanpuoleisia sulkuja ei tunnisteta ennen kuin OrQuery näyttää osaratkaisunsa.

esimerkiksi olisimme jättäneet const-määreen pois tai esitelleet toisen parametrin, ei NameQuery-ilmentymä olisi kumonnut kantaluokan aktiivista ilmentymää. Paluuarvon pitää olla myös sama yhdellä poikkeuksella: johdetun ilmentymän paluutyyppi voi olla julkisesti johdettu luokkatyyppi, joka on kantaluokan paluutyypin tyyppi. Jos esimerkiksi kantailmentymä palautti Query*:n, voi johdettu ilmentymä palauttaa NameQuery*:n. (Katsomme esimerkkiä, miksi saattaisimme tehdä näin, kun toteutamme clone()-funktiomme.) Tässä on NotQuery-esittelymme ja toteutuksemme print()-funktiosta:

```
class NotQuery : public Query {
public:
    virtual ostream& print( ostream &os ) const;
    // ...
};

ostream&
NotQuery::
print( ostream &os ) const
{
          os << " ! ";
          if ( _lparen )
        print_lparen( _lparen, os );

    _op->print( os );

    if ( _rparen )
        print_rparen( _rparen, os );
          return os;
}
```

print()-funktion käynnistys _op:n kautta on tietysti virtuaalinen käynnistys.

AndQuery- ja OrQuery-esittelyt ja toteutukset ovat itse asiassa toistensa kopioita. Näytämme sen vain AndQuery:n kohdalta:

```
class AndQuery : public Query {
public:
    virtual ostream& print( ostream &os ) const;
    // ...
};

ostream&
AndQuery::
print( ostream &os ) const
{
    if ( _lparen )
        print_lparen( _lparen, os );

    _lop->print( os );
    os << " && ";
    _rop->print( os );
```

```
    if ( _rparen )
        print_rparen( _rparen, os );

    return os;
}
```

Tämän virtuaalisen print()-funktion toteutus mahdollistaa, että voimme tulostaa minkä tahansa Query-alityypin ostream-virtaan tai mihin tahansa ostream-virrasta johdettuun stream-virtaan. Esimerkiksi:

```
cout << "The query request is ";
Query *pq = retrieveQuery();
pq->print( cout );
```

Vaikka tämä piirre on hyödyllinen, se on riittämätön. Lisäksi haluamme pystyä tulostamaan minkä tahansa Query:stä johdetun nykyisen tai tulevan luokkatyypin käyttäen iostream-tulostusoperaattoria. Esimerkiksi:

```
Query *pq = retrieveQuery();
cout << "The query request "
   << *pq
   << " generated the following results:\n";
```

Emme voi tehdä virtuaalista tulostusoperaattoria suoraan, koska tulostusoperaattorit ovat jo ostream-luokan jäseniä. Sen sijaan meidän pitää tehdä epäsuora virtuaalifunktio kuten seuraavassa:

```
inline ostream&
operator<<( ostream &os, const Query &q )
{
    // print()-funktion virtuaalinen käynnistys
    return q.print( os );
}
```

Kun kirjoitamme

```
AndQuery query;
// aseta kysely ...
cout << query << endl;
```

ostream-operaattorimme käynnistetään vuorostaan

```
q.print( os )
```

q:lla, joka on sidottu AndQuery-luokkaolioon query ja os on sidottu cout:iin. Jos sen sijaan kirjoitimme

```
NameQuery query2( "Salinger" );
cout << query2 << endl;
```

käynnistetään NameQuery-ilmentymän print(). Kutsu

```
Query *pquery = retrieveQuery();
```

```
cout << *pquery << endl;
```

käynnistää print()-ilmentymän, joka liittyy olioon, jota pquery osoittaa ohjelman suoritushetkellä.

17.5.2 Puhtaat virtuaalifunktiot

Kun haluamme tukea käyttäjän kyselyä, pääohjelmointitehtävämme on varsinaisten tyyppikohtaisten operaatioiden toteutus, jotka liittyvät jokaiseen kyselyoperaattoriin. Tämän toteuttamiseksi määrittelimme neljä todellista luokkatyyppiä: AndQuery, OrQuery jne. Suunnittelumme päätehtävä on kuitenkin kapseloida todellinen kysely tyyppiriippumattoman rajapinnan taakse. Tämä mahdollistaa, että voimme tehdä sovelluksemme ydinalueen koodilla, johon ei vaikuta tiettyjen tyyppien lisäys tai poisto.

Tämän toteuttamiseksi määrittelimme abstraktin Query-luokkatyyppimme. Emme ohjelmoi kyselytyyppejä, joita käyttäjä saattaa määrittää, vaan abstraktit toimenpiteet, joita voimme käyttää kaikkiin kyselytyyppeihin. Esimerkiksi:

```
void doit_and_bedone( vector< Query* > *pvec )
{
   vector<Query*>::iterator
      it = pvec->begin(),
      end_it = pvec->end();

   for ( ; it != end_it; ++it )
   {
      Query *pq = *it;
      cout << "processing " << *pq << endl;
      pq->eval();
      pq->display();
      delete pq;
   }
}
```

Teoriassa tämä tukee rajoittamatonta määrää tulevaisuudessa lisättäviä tyyppejä ilman, että tarvitsisi joko muuttaa tai kääntää uudelleen järjestelmän ydintä — edellyttäen, että abstrakti Query-kantaluokkamme julkinen rajapinta on riittävä tukemaan jokaista uutta kyselytyyppiä.

Tarkoituksemme on, kun teemme Query-luokkamme julkisen rajapinnan, että määrittelemme operaatiojoukon, joka on riittävä tukemaan kaikkia nykyisiä ja tulevia kyselytyyppejä. Käytännössä se on melko suuri lupaus eikä kukaan voi taata, että kaikkia tulevaisuuden kyselyoperaatioiden tyyppejä tuetaan. Niille tyypeille, jotka jo tiedämme, yleisen rajapinnan tekeminen on totta kai luvattavissa. Väitteisiin, että se tekisi vielä enemmän, tulisi suhtautua hieman skeptisesti.

Koska Query on abstrakti luokka, joka ei todellisuudessa esiinny sovelluksessamme, ei ole olemassa mielekästä toteutusta, jonka voisimme tehdä sen virtuaalifunktioille. Ne toimivat pikemminkin paikanpitäjinä, jotka johdetut alityypit korvaavat myöhemmin. Niitä itseään ei ole

aikomus käynnistää suoraan koskaan.

Kielellä on syntaktinen rakenne, jolla voidaan osoittaa, että virtuaalifunktio, jolla on rajapinta alityyppien kumottavaksi, mutta että sitä itseään ei pidä käynnistää virtuaalimekanismin kautta: *puhdas virtuaalifunktio*. Puhdas virtuaalifunktio esitellään seuraavasti:

```
class Query {
public:
   // esittelee aidon virtuaalifunktion
   virtual ostream& print( ostream&=cout ) const = 0;
   // ...
};
```

jossa funktion esittelyn jälkeen on sijoitus arvolla 0.

Luokan, joka sisältää (tai perii) yhden tai useamman puhtaan virtuaalisen funktion, kääntäjä tunnistaa abstraktiksi kantaluokaksi. Yritys luoda riippumaton luokkaolio abstraktista kantaluokasta johtaa käännösvirheeseen. (Samalla tavalla on virhe käynnistää puhdas virtuaalifunktio virtuaalimekanismin kautta.) Esimerkiksi:

```
// Query esittelee yhden tai useamman puhtaan virtuaalifunktion
// Ohjelmoija ei siksi saa luoda
// riippumattomia Query-luokkaolioita

// ok: Query-aliolio NameQuery:ssä
Query *pq = new NameQuery( "Nostromo" );

// virhe: new-lauseke varaa muistia Query-oliolle
Query *pq2 = new Query;
```

Abstrakti kantaluokka voi esiintyä vain alioliona siitä johdetuissa luokissa. Nämä ovat täsmälleen sellaisia asioita, joita haluamme Query-kantaluokallemme.

17.5.3 Virtuaalifunktion staattinen käynnistys

Kun käynnistämme virtuaalifunktion käyttäen luokan viittausalueoperaattoria, kumoamme virtuaalimekanismin, joka saa aikaan virtuaalifunktion ratkaisun staattisesti käännöksen aikana. Olettaen esimerkiksi, että olemme määritelleet `isA()`-virtuaalifunktion jokaiseen Query-hierarkian kantaluokkaan ja johdettuun luokkaan

```
Query *pquery = new NameQuery( "dumbo" );

// käynnistää isA()-funktion dynaamisesti virtuaalifunktion kautta
// NameQuery::isA()-ilmentymä käynnistetään
pquery->isA();

// käynnistää isA-funktion staattisesti käännöksen aikana
// Query::isA-ilmentymä käynnistetään
pquery->Query::isA();
```

Query::isA():n eksplisiittinen käynnistys ratkaistaan käännöksen aikana Query-kantaluokan ilmentymäksi, vaikka pquery sattuu osoittamaan NameQuery-oliota.

Miksi haluaisimme kumota virtuaalimekanismin? Usein tehokkuuden takia. Joskus on tarpeen käynnistää johdetun luokan virtuaalifunktiossa kantaluokan ilmentymä sellaisen operaation loppuunsaattamiseksi, joka on jaettu kantaluokkien ja johdettujen luokkien ilmentymien kesken. Esimerkiksi virtuaalinen display()-funktio Camera näyttää todennäköisesti tietoa, joka on yhteistä kaikille Camera-ilmentymille. display()-funktion PerspectiveCamera-ilmentymä näyttää tietoa, joka on samanlaista PerspectiveCamera-ilmentymälle. Sen sijaan, että monistettaisiin Camera-operaatiot PerspectiveCamera-toteutuksen display()-funktiossa, käynnistämme Camera-ilmentymän. Tiedämme tarkalleen ilmentymän, jonka haluamme käynnistää, joten ei ole tarvetta käydä läpi virtuaalimekanismia, jos voimme kumota sen. Lisäksi, jos Camera:n ilmentymä on esitelty välittömäksi, inline, sen käännöksenaikainen käynnistys johtaa välittömään laajentumiseen.

Tässä on toinen kuvaus siitä, milloin ehkä haluaisimme kumota virtuaalimekanismin. Siitä nähdään myös toinen näkökanta puhtaista virtuaalifunktiosta, sellainen, jota vasta-alkaneet C++-ohjelmoijat eivät usein huomaa heti.

print()-funktion AndQuery- ja OrQuery-ilmentymät ovat samanlaisia, paitsi literaalimerkkijono, joka edustaa operaattoria. Sen sijaan, että tekisimme kaksi ilmentymää, toteuttakaamme yksi ilmentymä, jonka nämä kaksi jakavat. Tämän toteuttamiseksi määrittelemme jälleen abstraktin BinaryQuery-kantaluokan, josta AndQuery ja OrQuery on johdettu. BinaryQuery määrittelee kaksi operandia ja lisäksi string-tietojäsenen, joka sisältää operaattorin arvon. Koska se on abstrakti luokka, esittelemme print():in puhtaaksi virtuaalifunktioksi.

```
class BinaryQuery : public Query {
public:
    BinaryQuery( Query *lop, Query *rop, string oper )
            : _lop(lop), _rop(rop), _oper(oper){}

        ~BinaryQuery() { delete _lop; delete _rop; }
        ostream &print( ostream& =cout ) const = 0;

protected:
        Query *_lop;
        Query *_rop;
        string _oper;
};
```

Tässä on ainoa BinaryQuery:n ilmentymä print()-funktiosta, jonka johdetut luokat, AndQuery ja OrQuery, voivat käynnistää:

```
inline ostream&
BinaryQuery::
print( ostream &os ) const
{
    if ( _lparen )
        print_lparen( _lparen, os );
```

```
    _lop->print( os );
    os << ' ' << _oper << ' ';
    _rop->print( os );

    if ( _rparen )
        print_rparen( _rparen, os );

    return os;
}
```

Hmm. Näyttää siltä, että olemme saattaneet itsemme paradoksiin. Toisaalta pidämme välttämättömänä esitellä tämä print()-ilmentymä puhtaana virtuaalifunktiona ilmaistaksemme BinaryQuery:n abstraktina kantaluokkana kääntäjälle. Olemme nyt taanneet, että BinaryQuery-luokan itsenäisiä olioita ei voi määritellä sovelluksessamme.

Toisaalta meille on välttämätöntä sekä määritellä BinaryQuery:n virtuaalinen print()-ilmentymä että käynnistää se AndQuery- ja OrQuery-luokkien olioiden kautta.

Kuten on tavallista selkeässä paradoksissa, meiltä puuttuu tietoa: puhdas virtuaalifunktio voidaan käynnistää staattisesti, vaikka sitä ei voida käynnistää virtuaalimekanismin kautta. Esimerkiksi:

```
inline ostream&
AndQuery::
print( ostream &os ) const
{
    // ok: pidätä virtuaalimekanismia
    // käynnistä BinaryQuery::print staattisesti
    BinaryQuery::print( os );
}
```

17.5.4 Virtuaalifunktiot ja oletusargumentit

Mietipä seuraavaa yksinkertaista luokkahierarkiaa:

```
#include <iostream>

class base {
public:
    virtual int foo( int ival = 1024 ) {
        cout << "base::foo() -- ival: " << ival << endl;
        return ival;
    }

    // ...
};

class derived : public base {
public:
    virtual int foo( int ival = 2048 ) {
```

```
            cout << "derived::foo() -- ival: " << ival << endl;
            return ival;
        }

        // ...
};
```

Luokan suunnittelijan tarkoitus on, että kantaluokan `foo()`-ilmentymälle, jos se käynnistetään ilman argumenttia, tulisi välittää oletusargumentti 1024. Esimerkiksi:

```
base b;
base *pb = &b;

// base::foo( int ) käynnistetään
// tarkoitus on, että sen tulisi palauttaa arvo 1024
pb->foo();
```

Samalla tavalla ohjelmoijan tarkoitus on, että johdetun luokan `foo()`-ilmentymälle, jos se käynnistetään ilman argumenttia, tulisi välittää oletusargumentti 2048. Esimerkiksi:

```
derived d;
base *pb = &d;

// derived::foo( int ) käynnistetään --
// tarkoitus on, että sen tulisi palauttaa arvo 2048
pb->foo();
```

Sattumalta tämä ei ole C++-virtuaalimekanismin normaalia käyttäytymistä. Tässä on esimerkiksi pieni ohjelma luokkahierarkiamme kokeiluun:

```
int main()
{
    derived *pd = new derived;
    base *pb = pd;

    int val = pb->foo();
    cout << "main() : val through base: "
        << val << endl;

    val = pd->foo();
    cout << "main() : val through derived: "
        << val << endl;
}
```

Kun ohjelma käännetään ja suoritetaan, se generoi seuraavan tulostuksen:

```
derived::foo() -- ival: 1024
main() : val through base: 1024
derived::foo() -- ival: 2048
main() : val through derived: 2048
```

Molemmissa käynnistyksissä foo():n johdetun luokan ilmentymä käynnistetään oikein. Tämä siksi, koska käynnistetty foo():n todellinen ilmentymä päätellään käännöksen aikana ja se perustuu todelliseen luokkatyyppiin, jota sekä pd että pb osoittavat. foo():lle välitettävää oletusargumenttia ei kuitenkaan päätellä suorituksen aikana; sen sijaan se päätellään käännöksen aikana ja se perustuu olion tyyppiin, jonka kautta funktio käynnistetään. Kun foo() käynnistetään pb:n kautta, päätellään oletusargumentti base::foo():n esittelystä, joka on 1024. Kun foo() käynnistetään pd:n kautta, päätellään oletusargumentti derived::foo():n esittelystä, joka on 2048.

Jos johdetun luokan ilmentymälle, kun se käynnistetään kantaluokan osoittimen tai viittauksen kautta, välitetään kantaluokan määrittämä oletusargumentti, niin miksi määrittää oletusargumentti johdetun luokan ilmentymään?

Voimme haluta erilaisen oletusargumentin, joka ei perustu tietyn foo():n alityypin toteutuksen käynnistykseen, vaan sen sijaan osoittimen tai viittauksen tyyppiin, jonka kautta funktio käynnistetään. Esimerkiksi arvot 1024 ja 2048 saattavat edustaa kuvakokoja. Jos haluamme tuottaa kuvan, jossa on vähemmän yksityiskohtia, käynnistämme foo():n kannan kautta. Jos haluamme hienompaa resoluutiota, käynnistämme foo():n johdetun kautta.

Mutta mitä, jos todella haluamme foo():lle välitettävän varsinaisen oletusargumentin, joka perustuu käynnistettävän funktion todelliseen ilmentymään? Valitettavasti virtuaalimekanismi ei tue tätä suoraan. Eräs ohjelmointiratkaisu on määrittää oletusargumentti, jonka voidaan havaita ilmaisevan, että käyttäjä ei ole välittänyt arvoa. Aiottu oletusargumentti sen sijaan esitellään paikalliseksi funktiolle ja käytetään, ellei eksplisiittistä arvoa välitetä. Esimerkiksi:

```
void
base::
foo( int ival = base_default_value )
{
    int real_default_value = 1024;

    if ( ival == base_default_value )
        ival =  real_default_value;

    // ...
}
```

jossa base_default_value on sovittu arvo koko hierarkialle ja läsnäolollaan ilmaisee, että käyttäjä ei ole antanut eksplisiittistä arvoa. Johdetun luokan ilmentymä toteutetaan samankaltaisella tavalla:

```
void
derived::
foo( int ival = base_default_value )
{
    int real_default_value = 2048;

    if ( ival == base_default_value )
        ival =  real_default_value;
```

```
    // …
}
```

17.5.5 Virtuaaliset tuhoajat

Seuraavassa funktiossa käytämme delete-lauseketta näin:

```
void doit_and_bedone ( vector< Query* > *pvec )
{
  // ...
  for ( ; it != end_it; ++it )
  {
      Query *pq = *it;
      // ...
      delete pq;
  }
}
```

Jotta funktio toimisi oikein, pitää käynnistää dynaamisen tyypin tuhoaja, johon pq osoittaa, kun käytetään delete-lauseketta. Jotta se tapahtuisi, pitää Query-luokan tuhoaja esitellä virtuaaliseksi:

```
class Query {
public:
  virtual ~Query() { delete _solution; }
  // ...
};
```

Jokaisen myöhemmin johdetun luokan tuhoajaa kohdellaan nyt automaattisesti virtuaalisena. doit_and_bedone() toimii oikein.

Tuhoajan toiminta periytymisessä on seuraava: johdetun luokan tuhoaja käynnistetään ensin. pq:n tapauksessa se on virtuaalifunktion kutsu. Päättymisvaiheessa lähimmän kantaluokan tuhoaja käynnistetään staattisesti — välittömäksi laajennettuna, jos se on sellaiseksi esitelty. Jos esimerkiksi pq osoittaa AndQuery-oliota

```
delete pq;
```

käynnistyy AndQuery-tuhoaja virtuaalimekanismin kautta. Kun se on tehty, käynnistetään BinaryQuery-tuhoaja staattisesti. Sen jälkeen käynnistetään Query-tuhoaja myös staattisesti.

Olkoon seuraava luokkahierarkia:

```
class Query {
public: // ...
protected:
  virtual ~Query();
  // ...
};

class NotQuery : public Query {
public:
```

```
    ~NotQuery();
    // ...
};
```

NotQuery-tuhoajan käsittelytaso on julkinen, kun se käynnistetään NotQuery-olion kautta, mutta suojattu, kun se käynnistetään Query-osoittimen tai -viittauksen kautta. Tämä tarkoittaa, että virtuaalifunktio olettaa luokkatyypin käsittelytason siitä, jonka kautta se on käynnistetty. Täten:

```
int main()
{
    Query *pq = new NotQuery;

    // ei sallittu: tuhoaja on suojattu
    delete pq;
}
```

Yleisenä peukalosääntönä suosittelemme, että jos luokkahierarkian juuren kantaluokka esittelee yhden tai useamman virtuaalifunktion, esittelisi se myös tuhoajansa virtuaaliseksi. Toisin kuin kantaluokan muodostajaa, ei kantaluokan tuhoajaa kuitenkaan tulisi yleensä tehdä suojatuksi.

17.5.6 Virtuaalifunktio eval()

Kyselyluokkahierarkian sydän on eval()-virtuaalifunktio (ja itse asiassa vähiten kiinnostava kielen piirteiden kannalta). Jälleen kerran ei abstraktille Query-luokalle ole mielekästä toteutusta, joten esittelemme sen puhtaaksi virtuaali-ilmentymäksi:

```
class Query {
public:
    virtual void eval() = 0;
    // ...
};
```

Nimen varsinainen arviointi tapahtuu sanojen sijaintikartan muodostamisen yhteydessä. Jos sana löytyy tekstistä, sen paikkavektori on mukana kartassa. Jos toteutuksessamme on paikkavektori mukana, se välitetään NameQuery-muodostajalle sanan nimen kanssa. NameQuery:n eval()-ilmentymälle ei ole mitään tekemistä.

Kuitenkaan emme voi sallia, että se perisi Query:n esittelemää puhdasta virtuaali-ilmentymää. Miksi? NameQuery on konkreettinen luokka, joka edustaa todellisia olioita sovelluksessamme. Jos se perisi Query:n puhtaan virtuaalifunktion, olisi NameQuery abstrakti luokka ja sitä estettäisiin luomasta NameQuery-luokkaoliota. Sen sijaan määrittelemme yksinkertaisesti eval():in tyhjäksi funktioksi:

```
class NameQuery : public Query {
public:
    virtual void eval() {}
    // ...
```

```
};
```

NotQuery löytää tekstistä jokaisen rivin, josta operandia ei löydy. Kaikki näiden rivien rivi- ja sarakeparit lisätään NotQuery-ilmentymään _loc. Tässä on toteutuksemme:

```
void NotQuery::eval()
{
    // varmistu, että operandi arvioidaan
    _op->eval();

    // _all_locs on tekstin kaikkien sijaintiparien vektori
    // se on NotQuery:n staattinen jäsen:
    // static const vector<location>* _all_locs
    vector< location >::const_iterator
            iter = _all_locs->begin(),
            iter_end = _all_locs->end();

    // hae rivijoukko, joissa operandi esiintyy
    set<short> *ps = _vec2set( _op->locations() );

    // jokaisesta rivistä, josta ei operandia löytynyt,
    // kopioi sijaintipari _loc:iin
    for ( ; iter != iter_end; ++iter )
    {
        if ( ! ps->count( (*iter).first ))
            _loc.push_back( *iter );
    }
}
```

Seuraavassa on Ei-kyselyn arvioinnin seuranta. Operandi esiintyy tekstin riveillä 0, 3 ja 5. (Muista, että sisäisesti indeksoimme tekstin string-vektorissa ja aloitamme laskennan arvosta nolla; kun näytämme rivinumerot käyttäjälle, aloitamme numerosta yksi.) Tästä syystä NotQuery:n arviointi luo paikkavektorin, johon tulevat rivien 1, 2 ja 4 rivi- ja sarakeparit. (Olemme muokanneet paikkavektoria minimoidaksemme näyttöä.)

```
==> ! daddy
daddy ( 3 ) lines match
display_location vector:
    first: 0    second: 8
    first: 3    second: 3
    first: 5    second: 5
! daddy ( 3 ) lines match
display_location vector:
    first: 1    second: 0
    first: 1    second: 1
    first: 1    second: 2
    ...
    first: 1    second: 10
    first: 2    second: 0
    first: 2    second: 1
```

```
...
first: 2      second: 12
first: 4      second: 0
first: 4      second: 1
...
first: 4      second: 12

Requested query:  ! daddy
( 2 ) when the wind blows through her hair, it looks almost alive,
( 3 ) like a fiery bird in flight. A beautiful fiery bird, he tells her,
( 5 ) she tells him, at the same time wanting him to tell her more.
```

OrQuery yhdistää kahden operandinsa paikkavektorit. Se käyttää hyödykseen geneeristä merge()-algoritmia. Jotta merge() pystyisi lajittelemaan rivi- ja sarakeparit, määrittelemme funktio-olion, joka päättelee, kumpi kahdesta rivi- ja sarakeparista on pienempi kuin toinen. Tässä toteutuksemme:

```
class less_than_pair {
public:
    bool operator()( location loc1, location loc2 )
    {
        return (( loc1.first < loc2.first ) ||
                ( loc1.first == loc2.first ) &&
                ( loc1.second < loc2.second ));
    }
};

void OrQuery::eval()
{
    // vertaile vasen ja oikea operandi
    _lop->eval();
    _rop->eval();

    // valmistaudu yhdistämään nämä kaksi paikkavektoria
    vector< location, allocator >::const_iterator
        riter = _rop->locations()->begin(),
        liter = _lop->locations()->begin(),
        riter_end = _rop->locations()->end(),
        liter_end = _lop->locations()->end();

    merge( liter, liter_end, riter, riter_end,
        inserter( _loc, _loc.begin() ),
        less_than_pair() );
}
```

Seuraavassa on Tai-kyselyn arvioinnin seurantaa, jossa näytämme jokaisen OrQuery-operandin paikkavektorin ja tuloksena olevan merge()-yhdistelyn. (Muista jälleen, että rivinumerot näytetään käyttäjälle ykkösestä alkaen, kun taas sisäisesti ne alkavat nollasta.)

```
==> fiery || untamed
fiery ( 1 ) lines match
display_location vector:
        first: 2        second: 2
        first: 2        second: 8

untamed ( 1 ) lines match
display_location vector:
        first: 3        second: 2

fiery || untamed ( 2 ) lines match
display_location vector:
        first: 2        second: 2
        first: 2        second: 8
        first: 3        second: 2

Requested query: fiery || untamed
( 3 ) like a fiery bird in flight. A beautiful fiery bird, he tells her,
( 4 ) magical but untamed. "Daddy, shush, there is no such thing,"
```

AndQuery-toteutus iteroi läpi kahden operandinsa paikkavektoreita etsien vierekkäisiä sanoja. Jokainen pari, jonka se löytää, lisätään _loc:iin. Sen toteutuksen päätyö on pitää kahden operandinsa paikat tahdissa niin, että voimme vertailla niiden vierekkäisyyttä.

```
void AndQuery::eval()
{
        // vertaile vasen ja oikea operandi
        _lop->eval();
        _rop->eval();

        // kahmaise iteraattorit
        vector< location, allocator >::const_iterator
                riter = _rop->locations()->begin(),
                liter = _lop->locations()->begin(),
                riter_end = _rop->locations()->end(),
                liter_end = _lop->locations()->end();


           // silmukoi läpi niin kauan, kuin molemmissa on elementtejä vertailua varten
        while ( liter != liter_end && riter != riter_end )
        {
                // niin kauan, kuin vasen rivinumero on suurempi kuin oikea
                while ( (*liter).first > (*riter).first ) {
                        ++riter;
                        if ( riter == riter_end ) return;
                }
```

```
            // niin kauan, kuin vasen rivinumero on pienempi kuin oikea
            while ( (*liter).first < (*riter).first )
            {
                // kun rivin viimeinen sana täsmää
                // ja toisen rivin ensimmäinen …
                // _max_col: yksilöi rivin viimeisen sanan

                if ( (*liter).first == (*riter).first-1 &&
                     (*riter).second == 0 &&
                     (*liter).second == (*_max_col)[ (*liter).first ] )
                {
                    _loc.push_back( *liter );
                    _loc.push_back( *riter );
                    ++riter;
                    if ( riter == riter_end ) return;
                }

                ++liter;
                if ( liter == liter_end ) return;
            }

            // niin kauan, kuin molemmat ovat samalla rivillä
            while ( (*liter).first == (*riter).first )
            {
                if ( (*liter).second+1 == ((*riter).second) )
                { // ok: an adjacent match
                    _loc.push_back( *liter ); ++liter;
                    _loc.push_back( *riter ); ++riter;
                }
                else
                if ( (*liter).second <= (*riter).second )
                    ++liter;
                else ++riter;

                if ( liter == liter_end || riter == riter_end )
                    return;
            }
        }
    }
```

Seuraavassa on Ja-kyselyn arvioinnin seuranta, jossa näytämme jokaisen AndQuery-operandin paikkavektorin ja lopullisen arvioinnin paikkavektorin. (Muista jälleen, että rivinumerot näytetään käyttäjälle alkaen yhdestä, kun taas sisäisesti ne alkavat nollasta.)

```
==> fiery && bird
fiery ( 1 ) lines match
display_location vector:
        first: 2        second: 2
        first: 2        second: 8
bird ( 1 ) lines match
display_location vector:
        first: 2        second: 3
        first: 2        second: 9
fiery && bird ( 1 ) lines match
display_location vector:
        first: 2        second: 2
        first: 2        second: 3
        first: 2        second: 8
        first: 2        second: 9

Requested query: fiery && bird
( 3 ) like a fiery bird in flight. A beautiful fiery bird, he tells her,
```

Lopuksi näytämme seurannan yhdistetystä Ja-Tai-kyselyn arvioinnista. Jokaisen välituloksen paikkavektori näytetään sekä myös lopullisen tuloksen paikkavektori.

```
==> fiery && ( bird || untamed )
fiery ( 1 ) lines match
display_location vector:
        first: 2        second: 2
        first: 2        second: 8
bird ( 1 ) lines match
display_location vector:
        first: 2        second: 3
        first: 2        second: 9
untamed ( 1 ) lines match
display_location vector:
        first: 3        second: 2
( bird || untamed )  ( 2 ) lines match
display_location vector:
        first: 2        second: 3
        first: 2        second: 9
        first: 3        second: 2
fiery &&  ( bird || untamed )  ( 1 ) lines match
display_location vector:
        first: 2        second: 2
        first: 2        second: 3
        first: 2        second: 8
        first: 2        second: 9
```

```
Requested query: fiery &&  ( bird || untamed )
( 3 ) like a fiery bird in flight. A beautiful fiery bird, he tells her,
```

17.5.7 Todellinen virtuaalinen new-operaattori

Kun meillä on osoitin konkreettiseen kyselyn alityyppiin, on turhaa varata samanlaista oliota keosta. Esimerkiksi:

```
NotQuery *pnq;
// aseta pnq ...

// new-lauseke käynnistää
// NotQuery-kopiointimuodostajan …
NotQuery *pnq2 = new NotQuery( *pnq );
```

Jos meillä on osoitin abstraktiin Query-luokkaan, samanlaisen olion varaaminen on huomattavasti turhempaa. Esimerkiksi:

```
 const Query *pq = pnq->op();
// kuinka tuplaamme pq:n?
```

Jos pystyisimme esittelemään new-operaattorista virtuaalisen ilmentymän, olisi ongelma ratkaistu. New-operaattorin oikea ilmentymä käynnistettäisiin automaattisesti. Valitettavasti new-operaattoria ei voi tehdä virtuaaliseksi, koska se on staattinen jäsenfunktio, jota käytetään muistin varaamiseen ennen luokkaolion muodostamista (katso kohdasta 15.8 aiheen käsittely).

Vaikka emme voi tehdä new-operaattoria virtuaaliseksi, voimme tehdä sijais-new-operaattorin, jolla voimme varata ja kopioida olioita keskusmuistiin. Tätä sijaista kutsutaan yleensä nimellä `clone()`:

```
class Query {
public:
    virtual Query *clone() = 0;
    // ...
};
```

Tässä on eräs mahdollinen toteutus NameQuery-ilmentymästämme:

```
class NameQuery : public Query {
public:
    virtual Query *clone()
        // käynnistää NameQuery-kopiointimuodostajan
        { return new NameQuery( *this ); }

    // ...
};
```

Tämä toimii täysin oikein, kun kohdeosoitintyyppi on `Query*` kuten seuraavassa:

```
Query *pq = new NameQuery( "Valery" );
Query *pq2 = pq->clone();
```

Se toimii hieman huonommin, kun kohdeosoitintyyppi on todellinen NameQuery*. Siinä tapauksessa vaaditaan, että teemme tyyppimuunnoksen takaisinpäin palautetusta Query*-osoittimesta NameQuery*-osoittimeksi:

```
NameQuery *pnq = new NameQuery( "Rilke" );
NameQuery *pnq2 =
    static_cast<NameQuery*>( pnq->clone() );
```

(Syy siihen, miksi tyyppimuunnos takaisinpäin on tarpeellinen, selitetään kohdassa 19.1.1)

Sanoimme aikaisemmin, että on yksi poikkeus vaatimukseen, että johdetun luokan paluutyypin pitää olla täsmälleen sama kuin kantaluokan ilmentymällä. Poikkeus koskee juuri tätä tapausta. Jos virtuaalifunktion kantailmentymä palauttaa luokkatyypin (tai osoittimen tai viittauksen luokkatyyppiin), saattaa johdettu ilmentymä palauttaa luokan, joka on julkisesti johdettu luokasta, jonka kantaluokan ilmentymä palautti (tai osoittimen tai viittauksen luokkatyyppiin).

```
class NameQuery : public Query {
public:
  virtual NameQuery *clone()
      { return new NameQuery( *this ); }

// ...
};
```

Nyt sekä pq2:n että pnq2:n alustus voidaan tehdä ilman pakotettua tyyppimuunnosta:

```
// Query *pq = new NameQuery( "Broch" );
Query *pq2 = pq->clone(); // ok

// NameQuery *pnq = new NameQuery( "Rilke" );
NameQuery *pnq2 = pnq->clone(); // ok
```

Tässä on NotQuery-kloonin toteutus:

```
class NotQuery : public Query {
public:
    virtual NotQuery *clone()
        { return new NotQuery( *this ); }

    // ...
};
```

AndQuery- ja OrQuery-ilmentymät on toteutettu samalla tavalla. Jotta nämä clone()-ilmentymien toteutukset onnistuisivat, pitää tehdä eksplisiittiset NotQuery-, AndQuery- ja OrQuery-kopiointimuodostajien ilmentymät. Teemme juuri sen kohdassa 17.6.

17.5.8 Virtuaalifunktiot, -muodostajat ja tuhoajat

Näimme kohdassa 17.4, että johdetun luokkaolion muodostajien käynnistysjärjestys on: ensin kantaluokan muodostaja ja sitten johdetun luokan muodostaja. Kun esimerkiksi määrittelemme NameQuery-luokkaolion kuten tässä

```
NameQuery poet( "Orlen" );
```

muodostajien käynnistysjärjestys on: Query ensin, NameQuery sitten.

Kun Query-kantaluokan muodostajaa suoritetaan, on poet:in NameQuery-osuus alustamatta. Itse asiassa poet ei ole vielä NameQuery; vain sen Query-aliolio on muodostettu.

Mitä tulisi tapahtua, jos kantaluokan muodostajassa käynnistetään virtuaalifunktio, josta sekä kantaluokan että johdetun luokan ilmentymä on määritelty? Mikä käynnistettäisiin? Jos olisi mahdollista, että virtuaalifunktion johdetun luokan ilmentymä käynnistettäisiin ja jos sen tulisi käsitellä jotakin johdetun luokan jäseniä, on käynnistyksen tulos muodollisesti tuntematon. Epävirallisesti ohjelma todennäköisesti romahtaa eli “kaatuu”.

Tämän estämiseksi virtuaali-ilmentymä, joka käynnistetään kantaluokan muodostajassa, on aina se virtuaalinen ilmentymä, joka on aktiivinen kantaluokassa. Itse asiassa kantaluokan muodostajan sisällä johdetun luokan olio on kantaluokkatyyppiä.

Sama pätee kantaluokan tuhoajan sisällä johdetun luokan oliolle: johdetun luokan osuus on myös tuntematon, mutta tällä kertaa se ei johdu siitä, että sitä ei ole muodostettu, vaan koska sitä on jo alettu tuhoata.

Harjoitus 17.12

NameQuery-oliossa selkein paikkavektorin sisäinen esitystapa on osoitin, joka alustetaan osoittimella, joka on tallennettu tekstin sijaintikarttaan. Tämä on myös kaikkein tehokkain, koska kopioimme yhden osoitteen sen sijaan, että kopioisimme vektorin jokaisen sijaintipaikkaparin. AndQuery-, OrQuery- ja NotQuery-luokkien pitää muodostaa paikkavektorinsa operandien arvioinnin perusteella. Kun jonkin tällaisen luokkaolion elinkaari päättyy, pitää siihen liittyvä paikkavektori hävittää. Kun NameQuery-olion elinkaari päättyy, paikkavektoria *ei saa* poistaa. Kuinka tallentaisimme paikkavektorin osoittimena Query-kantaluokkaan ja poistaisimme ilmentymät, jotka liittyvät AndQuery-, OrQuery- ja NotQuery-luokkaolioihin, mutta ei sitä, joka liittyy NameQuery-luokkaolioihin? (Huomaa, että emme saa lisätä lippua Query-luokkaan ilmaisemaan, poistetaanko osoitinta paikkavektoriin vai ei!)

Harjoitus 17.13

Mitä on pielessä seuraavassa luokkamäärittelyssä?

```
class AbstractObject {
public:
    ~AbstractObject();
    virtual void doit() = 0;
    // ...
};
```

Harjoitus 17.14

Miksi on niin, että kun on annettu

```
NameQuery nq( "Sneezy" );
Query q( nq );
Query *pq = &nq;
```

niin lauseke

```
pq->eval();
```

käynnistää eval():in NameQuery-ilmentymän, kun taas

```
q.eval();
```

käynnistää Query-ilmentymän?

Harjoitus 17.15

Mitkä Derived-luokan virtuaalifunktioiden uudelleenesittelyistä ovat virheellisiä?

```
(a) Base* Base::copy( Base* );
    Base* Derived::copy( Derived* );

(b) Base* Base::copy( Base* );
    Derived* Derived::copy( Base* );

(c) ostream& Base::print( int, ostream&=cout );
    ostream& Derived::print( int, ostream& );

(d) void Base::eval() const;
    void Derived::eval();
```

Harjoitus 17.16

Käytännössä ohjelmamme tuskin toimivat oikein, kun ensimmäisen kerran kokeilemme niitä ja kun kokeilemme niitä todellisella tiedolla. On usein hyödyllistä ottaa mukaan virheenkorjausstrategiaa luokkien suunnitteluun. Toteuta Query-luokkahierarkiaamme debug()-virtuaalifunktio, joka näyttää erikseen jokaisen luokan tietojäsenet. Tue yksityiskohtien kontrollia tasolla (a) argumentti debug()-funktioon ja (b) luokan tietojäsen. (Jälkimmäinen mahdollistaa

yksittäisten luokkaolioiden ottaa käyttöön ja pois käytöstä virheenetsintätietojen näyttämisen.)

Harjoitus 17.17

Mitkä ovat todennäköisiä virheitä seuraavassa periytymishierarkiassa?

```
class Object {
public:
    virtual void doit() = 0;
    // ...
protected:
    virtual ~Object();
};

class MyObject : public Object {
public:
    MyObject( string isA );
    string isA() const;
protected:
    string _isA;
};
```

17.6 Jäsenittäinen alustaminen ja sijoittaminen

Eräs velvollisuutemme luokkia suunnitellessamme on varmistua, että luokka käyttäytyy oikein ja tehokkaasti jäsenittäisessä alustamisessa (esitelty kohdassa 14.6) ja jäsenittäisessä sijoituksessa (esitelty kohdassa 14.7). Tässä kohdassa mietimme näitä operaatioita periytymisen yhteydessä.

Tähän mennessä emme ole tehneet yhtään jäsenittäisen alustuksen eksplisiittistä käsittelyä. Käykäämme läpi, mitä tapahtuu oletusarvoisesti Query-luokkahierarkiassamme.

Abstrakti Query-kantaluokka määrittelee kolme ei-staattista tietojäsentä:

```
class Query {
public: // ...
protected:
    int _paren;
    set<short> *_solution;
    vector<location> _loc;

    // ...
};
```

Jos _solution on asetettu, osoittaa se joukkoa, joka on varattu keskusmuistista _vec2set()-jäsenfunktiolla. Query-tuhoaja käyttää delete-lauseketta _solution:in tuhoamiseen.

Query-luokkaan pitää tehdä sekä eksplisiittinen kopiointimuodostaja että eksplisiittinen sijoitusoperaattori. (Ellei tämä ole selvää sinulle, katso pikaisesti uudelleen kohta 14.6). Kuitenkin, ennen kuin teemme nämä, käykäämme läpi oletusarvoinen jäsenittäinen kopiointi, kun ne

puuttuvat.

Johdettu NameQuery-luokka sisältää string-jäsenluokkaolion ja Query-kantaluokan alioliion. Olkoon seuraavassa NameQuery-olio, folk:

```
NameQuery folk( "folk" );
```

Kun music-olio alustetaan folk-oliolla

```
NameQuery music = folk;
```

saa se aikaan seuraavaa:

1. Kääntäjä tarkistaa nähdäkseen, onko NameQuery-luokkaan määritelty eksplisiittinen ilmentymä kopiointimuodostajasta. Ei ole. Tästä syystä kääntäjä varautuu käyttämään oletusarvoista jäsenittäistä alustusta.
2. Kääntäjä tarkistaa nähdäkseen, sisältääkö NameQuery-luokka yhtään kantaluokan alioliota. Kyllä, se sisältää Query-kantaluokan aliolion.
3. Kääntäjä tarkistaa nähdäkseen, onko Query-kantaluokkaan määritelty eksplisiittinen ilmentymä kopiointimuodostajasta. Ei ole. Tästä syystä kääntäjä varautuu käyttämään oletusarvoista jäsenittäistä alustusta.
4. Kääntäjä tarkistaa nähdäkseen, sisältääkö Query-luokka yhtään kantaluokan alioliota. Ei sisällä.
5. Kääntäjä tutkii Query-luokan jokaisen ei-staattisen jäsenen esittelyjärjestyksessä. Jos jäsen on luokaton olio kuten _paren ja _solution, se alustaa music-jäsenolion folk-jäsenen arvolla. Jos jäsen on luokkaolio kuten _loc, se käyttää kohtaa 1 rekursiivisesti. Kyllä, vektoriluokassa on määritelty eksplisiittinen ilmentymä kopiointimuodostajasta. Kopiointimuodostaja käynnistetään music._loc-olion alustamiseen folk._loc-oliolla.
6. Kääntäjä tutkii seuraavaksi NameQuery-luokan jokaisen ei-staattisen jäsenen esittelyjärjestyksessä. Havaitaan, että string-luokkaoliossa on eksplisiittinen kopiointimuodostaja. Se käynnistetään music._name-olion alustamiseen folk._name-oliolla.

music-olion oletusalustus folk-oliolla on nyt valmis. Se toimii hyvin, paitsi _solution:in oletuskopioinnissa, sillä jos se sallitaan, se saa todennäköisesti aikaan ohjelmavirheen. Kumoamme oletuskäsittelyn tekemällä Query-luokalle eksplisiittisen kopiointimuodostajan. Eräs ratkaisu on kopioida koko ratkaisujoukko kuten seuraavassa:

```
Query::Query( const Query &rhs )
        : _loc( rhs._loc ), _paren(rhs._paren)
{
    if ( rhs._solution )
    {
        _solution = new set<short>;
        set<short>::iterator
                    it = rhs._solution->begin(),
                    end_it = rhs._solution->end();
```

```
            for ( ; it != end_it; ++it )
                _solution->insert( *it );
        }
        else _solution = 0;
    }
```

Koska kuitenkin toteutuksemme päätös on laskea ratkaisujoukko vaadittaessa, ei ole olemassa pakkoa kopioida sitä nyt. Kopiointimuodostajamme tarkoitus on estää oletuskopiointi. On riittävä, kun alustamme _solution:in arvolla 0:

```
Query::Query( const Query &rhs )
        : _loc( rhs._loc ),
        _paren(rhs._paren), _solution( 0 )
{}
```

music:in ja folk:in alustaminen noudattaa samoja kohtia 1 ja 2. Kohdassa 3 huomataan nyt, että Query:yn on määritelty eksplisiittinen kopiointimuodostaja. Se käynnistetään. Kohtia 4 ja 5 ei enää suoriteta. Kohta 6 suoritetaan kuten ennen.

Jälleen kerran: music:in ja folk:in oletusarvoinen jäsenittäinen alustaminen on valmis. Se toimii hyvin. NameQuery:yn ei tarvitse tehdä eksplisiittistä kopiointimuodostajaa.

Johdettu NotQuery-luokka sisältää Query-kantaluokan aliolion ja Query*-tietojäsenen _op, joka osoittaa keskusmuistista varattua operandiaan. NotQuery-tuhoaja käyttää delete-lauseketta operandiin.

NotQuery-luokka ei turvallisesti salli oletusarvoista jäsenittäistä alustusta sen _op-jäsenelle, joten eksplisiittinen kopiointimuodostaja pitää tehdä. Se toteutus käyttää hyväkseen virtuaalista clone()-funktiota, joka määriteltiin edellisessä kohdassa.

```
inline NotQuery::
NotQuery( const NotQuery &rhs )
        // käynnistää: Query::Query( const Query &rhs )
        : Query( rhs )
    { _op = rhs._op->clone(); }
```

Yhden NotQuery-olion jäsenittäinen alustus toisella saa aikaan seuraavat kaksi vaihetta:

1. Kääntäjä tarkistaa, onko NotQuery-luokkaan määritelty eksplisiittinen ilmentymä kopiointimuodostajasta. On.
2. NotQuery-kopiointimuodostaja käynnistetään suorittamaan jäsenittäinen alustaminen.

Se on siinä. NotQuery:n kopiointimuodostajan on velvollisuus toteuttaa sekä kantaluokan aliolion että ei-staattisten tietojäsenten oikea alustaminen. AndQuery- ja OrQuery-ilmentymät ovat samanlaisia kuin NotQuery, ja ne jätetäänkin harjoitukseksi käyttäjälle.

Jäsenittäinen sijoitus on samanlaista kuin jäsenittäinen alustaminen. Jos eksplisiittinen kopioinnin sijoitusoperaattori on mukana, se käynnistetään luokkaolion sijoittamiseksi toiseen. Muussa tapauksessa käytetään jäsenittäistä sijoittamista.

Jos kantaluokka on mukana, kantaluokan aliolioon sijoitetaan jäsenittäin ensin. Jos kantaluokassa on eksplisiittinen kopioinnin sijoitusoperaattori, se käynnistetään. Muussa tapaukses-

sa käytetään oletusarvoista jäsenittäistä sijoittamista rekursiivisesti kantaluokkiin ja kantaluokkien aliolioiden jäseniin.

Jokainen ei-staattinen tietojäsen tutkitaan vuorollaan esittelyjärjestyksessä. Jos se on luokaton tyyppi, oikeanpuoleinen ilmentymä kopioidaan vasemmanpuoleiseen. Jos se on luokkatyyppi ja luokkaan on määritelty eksplisiittinen kopioinnin sijoitusoperaattori, operaattori käynnistetään. Muussa tapauksessa käytetään oletusarvoista jäsenittäistä sijoittamista rekursiivisesti kantaluokkiin ja jäsenluokkaolioiden jäseniin.

Tässä on Query:n kopioinnin sijoitusoperaattori. Jälleen kerran: ratkaisujoukkoa ei tarvitse kopioida tässä vaiheessa, vaan yksinkertaisesti estää sen oletuskopiointi:

```
Query&
Query::
operator=( const Query &rhs )
{
        // estä sijoitus itseensä
        if ( &rhs != this )
        {
            _paren = rhs._paren;
            _loc = rhs._loc;
            delete _solution;
            _solution = 0;
        }

        return *this;
}
```

NameQuery ei vaadi eksplisiittistä kopioinnin sijoitusoperaattoria. Yhden NameQuery-olion sijoitus toiseen johtaa seuraaviin kahteen vaiheeseen:

1. Query:n eksplisiittinen kopioinnin sijoitusoperaattori käynnistetään sijoittamaan kahden NameQuery-olion Query-alioliot.
2. Eksplisiittinen string-luokan kopioinnin sijoitusoperaattori käynnistetään sijoittamaan kahden NameQuery-olion string-jäsenluokkaoliot.

Emme voi tehdä parempaa eksplisiittistä kopioinnin sijoitusoperaattoria NameQuery:lle. Oletusarvoinen jäsenittäinen sijoittaminen on riittävä.

NotQuery, OrQuery ja AndQuery vaativat jokainen eksplisiittisen kopioinnin sijoitusoperaattorin, jolla kopioidaan niiden vastaavat operandit turvallisesti. Tässä on NotQuery-ilmentymä:

```
inline NotQuery&
NotQuery::
operator=( const NotQuery &rhs )
{
    // estä sijoitus itseensä
    if ( &rhs != this )
    {
        // käynnistä Query:n kopioinnin sijoitusoperaattori
```

```
            this->Query::operator=( rhs );

            // kopioi operandi
            _op = rhs._op->clone();
        }

        return *this;
    }
```

Toisin kuin kopiointimuodostajan yhteydessä, kopioinnin sijoitusoperaattorissa ei ole olemassa erityistä osaa, jonka kautta voitaisiin käynnistää kantaluokan kopioinnin sijoitusoperaattori. On olemassa kaksi syntaktista muotoa sen käynnistämiseen: aikaisemmin kuvaamamme eksplisiittinen käynnistys ja eksplisiittinen tyyppimuunnos kuten seuraavassa:

```
(*static_cast<Query*>(this)) = rhs;
```

AndQuery:n ja OrQuery:n kopioinnin sijoitusilmentymät ovat samanlaisia, ja se jätetäänkin harjoituksiksi.

Tässä on pieni testiohjelma, jolla toteutustamme voidaan kokeilla — järkevä tarkistus, että se todella toimii. Yksinkertaisesti luomme tai kopioimme olion ja sitten tulostamme sen arvot.

```
#include "Query.h"

int
main()
{
    NameQuery nm( "alice" );
    NameQuery nm2( "emma" );

    NotQuery nq1( &nm );
    cout << "notQuery 1: " << nq1 << endl;

    NotQuery nq2( nq1 );
    cout << "notQuery 2: " << nq2 << endl;

    NotQuery nq3( &nm2 );
    cout << "notQuery 3: " << nq3 << endl;

    nq3 = nq2;
    cout << "notQuery 3 assigned nq2: " << nq3 << endl;

    AndQuery aq( &nq1, &nm2 );
    cout << "AndQuery : " << aq << endl;

    AndQuery aq2( aq );
    cout << "AndQuery 2: " << aq2 << endl;

    AndQuery aq3( &nm, &nm2 );
    cout << "AndQuery 3: " << aq3 << endl;
```

```
    aq2 = aq3;
    cout << "AndQuery 2 after assign: " << aq2 << endl;
}
```

Kun ohjelma käännetään ja suoritetaan, se generoi seuraavan tulostuksen:

```
notQuery 1:  ! alice
notQuery 2:  ! alice
notQuery 3:  ! emma
notQuery 3 assigned nq2:  ! alice
AndQuery :  ! alice && emma
AndQuery 2:  ! alice && emma
AndQuery 3: alice && emma
AndQuery 2 after assign: alice && emma
```

Harjoitus 17.18

Toteuta AndQuery:n ja OrQuery:n kopiointimuodostajat.

Harjoitus 17.19

Toteuta AndQuery:n ja OrQuery:n kopioinnin sijoitusoperaattorit.

Harjoitus 17.20

Mitkä ovat sen todennäköisimmät oireet, että luokkaan vaaditaan eksplisiittinen kopiointimuodostaja ja kopioinnin sijoitusoperaattori?

17.7 UserQuery-luokka käyttäjän kyselyiden hallintaan

Kun tehdään kysely kuten

```
fiery && ( bird || potato )
```

on tehtävämme muodostaa vastaava Query-hierarkia:

```
AndQuery
    NameQuery( "fiery" )
    OrQuery
        NameQuery( "bird" )
        NameQuery( "potato" )
```

Kysymys kuuluu, miten voimme parhaiten tehdä tämän? Kyselyn arvioinnin prosessi muistuttaa äärellistä tilakonetta (*finite state machine*). Aloitamme tyhjästä tilasta ja siirrymme kyselyn jokaisella elementillä tilasta toiseen, kunnes koko kysely on arvioitu. Toteutuksemme sydän on suuri switch-lause operaatiossa, jonka nimi on eval_query(). Kyselyn jokainen sana luetaan vuorollaan string-vektorista ja testataan jokaista mahdollista arvoa vastaan, jonka kysely voi sisältää:

```
vector<string >::iterator
    it    = user_query->begin(),
```

```
        end_it = user_query->end();

for ( ; it != end_it; ++it )
        switch( evalQueryString( *it ))
        {
            case WORD:
                evalWord( *it );
                break;

            case AND:
                evalAnd();
                break;

            case OR:
                evalOr();
                break;

            case NOT:
                evalNot();
                break;

            case LPAREN:
                ++_paren;
                ++_lparenOn;
                break;

            case RPAREN:
                --_paren;
                ++_rparenOn
                evalRParen();
                break;
          }
```

Viisi eval-operaatiota (evalWord(), evalAnd(), evalOr(), evalNot() ja evalRParen()) rakentavat varsinaisen Query-hierarkian. Ennen kuin katsotaan tarkemmin niiden toteutukset, mietitäänpä ensin niiden järjestely.

Eräs strategia on määritellä jokaisesta yksilöllinen funktio kuten alun perin teimme luvussa 6 tekstin kyselyrutiineille. Käyttäjän kysely ja johdetut Query-alityypit edustavat riippumatonta tietoa funktioille, joilla funktiot operoivat. Tämä edustaa proseduraalista ohjelmointimallia; sellaista, jota emme tavoittele.

Kuten teimme kohdassa 6.14, kun esittelimme TextQuery-luokan, johon kapseloimme luvun 6 operaatiomme ja tiedot, haluamme tässä esitellä UserQuery-luokan, johon kapseloimme ja jossa hallitsemme näitä operaatioita ja tietoja.

Yksi tietojäsen on string-vektori, joka tulee sisältämään käyttäjän varsinaisen kyselyn. Toinen tietojäsen on osoitintyyppi Query*, joka osoittaa kyselyn hierarkkiseen esitystapaan, joka on rakennettu eval_query():yn. Kolme lisäjäsentä on määritelty sulkujen käsittelyyn: _paren auttaa vaihtamaan operaattorien arvioinnin oletussidontajärjestystä (annamme esimerkin hetken ku-

luttua) ja _lparenOn sekä _rparenOn pitävät yllä nykyisen kyselysolmun sulkujen lukumäärää ja laatua (näimme näiden käyttöä kohdassa 17.5.1, kun käsittelimme virtuaalista print()-funktiota).

Näiden viiden jäsenen lisäksi tarvitsemme kaksi lisää. Mietipä seuraavaa kyselyä:

```
fiery || untamed
```

Varsinainen päämäärämme on esittää tämä kysely seuraavana OrQuery-oliona:

```
OrQuery
    NameQuery( "fiery" )
    NameQuery( "untamed" )
```

Kyselyn käsittelyjärjestys on kuitenkin hieman ongelmallinen. Ensiksi määrittelemme NameQuery:n, mutta emme ole vielä määritelleet OrQuery:ä, johon lisätä se. Se, mitä tarvitsemme, on talletuspaikka hetkeksi NameQuery-oliolle myöhempää hakua varten.

Perinteinen tietorakenne operaatiolle “jonkin sijoittaminen jonnekin myöhempää hakua varten” on pino. Sijoitamme NameQuery-oliomme pinoon. Kun seuraavaksi kohtaamme OrQuery-operaattorin, haemme NameQuery-oliomme ja välitämme sen OrQuery:n vasemmanpuoleisena operandina.

Kun se on toteutettu, mitä meidän tulisi tehdä OrQuery-oliolle? OrQuery-oliomme on tässä vaiheessa epätäydellinen. Siltä puuttuu oikeanpuoleinen operandi. Meidän pitää laittaa se sivuun, kunnes sen oikeanpuoleinen operandi on käytettävissä.

Voimme laittaa sen samaan pinoon, johon sijoitamme NameQuery-olion. OrQuery-olio edustaa kuitenkin erilaista käsittelytilaa: se on keskeneräinen operaattori. Sen sijaan pidämme parempana määritellä kaksi pinoa: yksi pino sisältää oliot, jotka ovat valmiita yhdistetyn kyselyn operandeja (Tähän sijoitamme NameQuery-olion ja annamme tälle pinolle nimen _query_stack.) ja toinen pino sisältää keskeneräiset operaattorit, joilta puuttuu oikeanpuoleinen operandi. Ajattelemme tämän pinon sisältävän nykyisen operaation loppuunsaattamista, joten annamme sille nimen _current_op. Tähän sijoitamme OrQuery-olion. Kun määrittelemme toisen NameQuery-olion, haemme OrQuery-olion _current_op-pinosta ja lisäämme NameQuery-olion sen toiseksi operandiksi. OrQuery-olio on nyt valmis. Työnnämme sen _query_stack-pinoon.

Kun käyttäjän kyselyn käsittely on valmis ja jos kaikki on mennyt hyvin, on _current_op tyhjä ja _query_stack-pinon tulisi sisältää yksi olio. Tämä olio on käyttäjän kyselyn koko esitystapa. Esimerkissämme se on OrQuery-olio.

Jotta voisimme nähdä, kuinka tämä toimii, käykäämme läpi muutama todellinen kysely. Ensimmäinen esimerkki on yksinkertainen NotQuery:

```
! daddy
```

Seuraavassa on kyselyn varsinaisen käsittelyn seuranta. Lopullinen olio _query_stack-pinossa on NotQuery-olio:

```
evalNot() : incomplete!
        push on _current_op (size == 1 )
evalWord() : daddy
        pop _current_op : NotQuery
        add operand: WordQuery : NotQuery complete!
```

```
        push NotQuery on _query stack
```

Sisennetty teksti eval-operaation alla ilmaisee suoritettua operaatiota. Toinen esimerkkimme, joka on yhdistetty OrQuery, kuvaa molempia tilanteita. Siinä kuvataan myös valmiin operaattorin työntö _query_stack-pinoon:

```
==> fiery || untamed || shyly

evalWord() : fiery
        push word on _query stack
evalOr() : incomplete!
        pop _query_stack : fiery
        add operand : WordQuery : OrQuery incomplete!
        push OrQuery on _current_op (size == 1 )
evalWord() : untamed
        pop _current_op : OrQuery
        add operand: WordQuery : OrQuery complete!
        push OrQuery on _query stack
evalOr() : incomplete!
        pop _query_stack : OrQuery
        add operand : OrQuery : OrQuery incomplete!
        push OrQuery on _current_op (size == 1 )
evalWord() : shyly
        pop _current_op : OrQuery
        add operand: WordQuery : OrQuery complete!
        push OrQuery on _query stack
```

Viimeinen esimerkkimme kuvaa sekä yhdistettyä kyselyä että sulkujen käyttöä arviointijärjestyksen vaihtamiseen:

```
==> fiery && ( bird || untamed )

evalWord() : fiery
        push word on _query stack
evalAnd() : incomplete!
        pop _query_stack : fiery
        add operand : WordQuery : AndQuery incomplete!
        push AndQuery on _current_op (size == 1 )
evalWord() : bird
        _paren is set to 1
        push word on _query stack
evalOr() : incomplete!
        pop _query_stack : bird
        add operand : WordQuery : OrQuery incomplete!
        push OrQuery on _current_op (size == 2 )
evalWord() : untamed
        pop _current_op : OrQuery
        add operand: WordQuery : OrQuery complete!
        push OrQuery on _query stack
evalRParen() :
```

```
_paren: 0 _curent_op.size(): 1
pop _query_stack : OrQuery
pop _current_op: AndQuery
add operand : OrQuery : AndQuery complete!
push AndQuery on _query_stack
```

Tekstinkyselyjärjestelmämme toteutus muodostuu kolmesta komponentista: (1) oliopohjaisesta TextQuery-luokasta, joka tekee varsinaisen tekstin käsittelyn (jälleen: se on kerrottu tarkemmin kohdassa 6.14); (2) oliokeskeisestä Query-luokkahierarkiasta, joka edustaa ja arvioi jokaisen käyttäjän kyselyn ja (3) oliopohjaisesta UserQuery-luokasta, joka edustaa äärellistä tilakonetta, jolla muodostetaan Query-hierarkia.

Tähän mennessä olemme toteuttaneet nämä kolme komponenttia suurin piirtein riippumattomasti toisistaan ilman konflikteja. Valitettavasti näin ei enää ole asianlaita. Huomaatko ongelman? Query-luokkahierarkia ei tue olion muodostamisvaatimuksia, joita juuri läpikäymämme UserQuery-toteutus sille esittää!

1. AndQuery-, OrQuery- ja NotQuery-luokat vaativat nykyisellään, että niiden vastaavat operandit ovat mukana jokaisen olion määrittelyhetkellä. Käsittelymme kuitenkin vaatii, että määrittelemme keskeneräisen olion.
2. Käsittelymme vaatii, että lisäämme myöhemmin operandin AndQuery-, OrQuery- ja NotQuery-luokkiin. Lisäksi tämän pitää olla virtuaalinen operaatio. Operandi pitää lisätä `Query*`-osoittimen kautta, joka on työnnetty `_current_op`-pinoon. Operandin lisääminen on kuitenkin tyyppiriippuvaista ja perustuu siihen, onko se unaarioperaatio (eli yksioperandinen kuten OrQuery) vai binäärioperaatio (eli kaksioperandinen kuten OrQuery tai AndQuery). Kuten olemme Query-luokkahierarkiamme määritelleet, ei siinä ole tätä operaatiota.

Se, mitä tapahtui, on, että analysointimme johti rajapintaan, joka eroaa suunnitelmamme varsinaisesta toteutuksesta. Se ei johdu siitä, että analyysi olisi väärin, vaan että se on yksinkertaisesti keskeneräinen. Tämä on enemmän tai vähemmän ongelma vaiheessa, jossa analyysin, suunnittelun ja toteutuksen vaiheet ovat erillisiä ja nähdään peräkkäisenä *vesiputouksena* ilman palautteen ja tarkistamisen mahdollisuutta. Pohjimmiltaan on hyväksyttävä tosiasia, että emme voi ajatella ja ennakoida kaikkea. Vaikeus on erottaa — oman ja johdon mielessä — monimutkaisen prosessin väistämättömät virheet ja yksilöiden ajattelemattomuudessa tai ajanpuutteen takia tehdyt virheet.

Tässä tapauksessa meidän pitää joko itse palata muokkaamaan Query-luokkahierarkiaa tai neuvotella, että muutokset tulee tehtyä. Tehottomassa organisaatiossa vastasyytteet viivyttävät toteutusta ja projekteista tulee vähitellen kunnianhimottomia ja byrokraattisempia. Meidän tapauksessamme tekstin kirjoittajina yksinkertaisesti pyörittelemme hieman koodia ja muokkaamme alityyppien muodostajia ja lisäämme virtuaalisen `add_op()`-jäsenfunktion, joka tukee operandien lisäystä operaattoriin sen jälkeen, kun se on määritelty (näemme sen käyttöä hetken kuluttua, kun katsomme `evalRParen()`- ja `evalWord()`-operaatioita).

17.7.1 UserQuery-luokan määrittely

UserQuery-olio voidaan alustaa joko osoittimella string-vektoriin, joka edustaa käyttäjän kyselyä tai myöhemmin välittää käyttäjän kyselyn osoite query()-jäsenfunktion kautta. Tämä mahdollistaa yhden kyselyolion käytön useisiin käyttäjän kyselyihin. Query-luokkahierarkian varsinainen muodostaminen toteutetaan eval_query()-operaatiolla. Esimerkiksi:

```
// määrittele ilmentymä ilman vastaavaa käyttäjäkyselyä
UserQuery user_query;

string text;
vector<string> query_text;

// pyöritä läpi jokainen käyttäjän pyyntö
do {
        while( cin  >> text )
                query_text.push_back( text );

        // välitä kysely UserQuery-oliolle
        user_query.query( &query_text );

        // arvioi kysely ja paluu
        // Query*-hierarkian juuri …
        Query *query = user_query.eval_query();
}
while ( /* käyttäjä-haluaa-jatkaa-tekstin-kyselyä */ );
```

Tässä on määritelmämme UserQuery-luokasta:

```
#ifndef USER_QUERY_H
#define USER_QUERY_H

#include <string>
#include <vector>
#include <map>
#include <stack>

typedef pair<short,short>          location;
typedef vector<location,allocator>  loc;

#include "Query.h"

class UserQuery {
public:
    UserQuery( vector< string,allocator > *pquery = 0 )
                : _query( pquery ), _eval( 0 ), _paren( 0 ) {}

    Query *eval_query();    // muodostaa hierarkian
    void   query( vector< string,allocator > *pq );
    void   displayQuery();
```

```
    static void
        word_map( map<string,loc*,less<string>,allocator> *pwm )
        { if ( !_word_map ) _word_map = pwm; }

private:
    enum QueryType { WORD = 1, AND, OR, NOT, RPAREN, LPAREN };

    QueryType evalQueryString( const string &query );
    void    evalWord( const string &query );
    void    evalAnd();
    void    evalOr();
    void    evalNot();
    void    evalRParen();
    bool    integrity_check();

    int                     _paren;
    Query       *_eval;
    vector<string> *_query;

    stack<Query*,vector<Query*> > _query_stack;
    stack<Query*,vector<Query*> > _current_op;

    static short _lparenOn, _rparenOn;
    static map<string,loc*,less<string>,allocator>
            *_word_map;
};

#endif
```

Huomaa, että esittelimme kaksi pinoamme niin, että ne sisältävät elementtejä, jotka ovat osoittimia Query:yn eivätkä itse Query-olioita. Vaikka molemmat toteutukset saavat sovelluksen toimimaan oikein, olioiden tallentaminen suoraan on huomattavan tehotonta: jokainen olio (ja sen operandit) pitää kopioida jäsenittäin pinoon (muista, että clone():n virtuaalikäynnistys kopioi jokaisen operandin jäsenittäin) ja sitten myöhemmin tuhota. Ellemme todella muokkaa säiliöön sijoittamiamme luokkaolioita, on niiden tallentaminen osoittimella merkittävästi tehokkaampaa.

Tässä ovat ne muutamat eval-operaatiot. Olemme määritelleet jokaisen välittömäksi. evalAnd()- ja evalOr()-operaatiot suorittavat seuraavat vaiheet: kumpikin vetävät _query_stack-pinosta (tähän menee kaksi operaatiota vakiokirjaston pinoluokalta; muista: top() elementin hakuun ja pop() sen poistoon pinosta). Jompikumpi varaa joko AndQuery- tai OrQuery-olion keosta, jolle välitetään olio, joka on haettu _query_stack-pinosta. Kumpikin välittää AndQuery- tai OrQuery-oliolle kaikkien vasemmanpuoleisten ja oikeanpuoleisten sulkujen lukumäärän, jotka operaattori tarvitsee itsensä tulostamiseen. Lopuksi jokainen työntää keskeneräisen operaattorin _current_op-pinoon:

```
    inline void
```

```
UserQuery::
evalAnd()
{
    Query *pop   = _query_stack.top(); _query_stack.pop();
    AndQuery *pq = new AndQuery( pop );

    if ( _lparenOn )
        {  pq->lparen( _lparenOn ); _lparenOn = 0; }

    if ( _rparenOn )
        {  pq->rparen( _rparenOn ); _rparenOn = 0; }

    _current_op.push( pq );
}

inline void
UserQuery::
evalOr()
{
    Query *pop  = _query_stack.top(); _query_stack.pop();
    OrQuery *pq = new OrQuery( pop );

    if ( _lparenOn )
        {  pq->lparen( _lparenOn ); _lparenOn = 0; }

    if ( _rparenOn )
        {  pq->rparen( _rparenOn ); _rparenOn = 0; }

    _current_op.push( pq );
}
```

evalNot()-operaation vaiheet ovat seuraavat: se varaa uuden NotQuery-olion keosta, välittää NotQuery-oliolle kaikkien vasemmanpuoleisten ja oikeanpuoleisten sulkujen lukumäärän, jotka operaattori tarvitsee itsensä näyttämiseen. Lopuksi se työntää keskeneräisen operaattorin _current_op-pinoon:

```
inline void
UserQuery::
evalNot()
{
    NotQuery *pq = new NotQuery;

    if ( _lparenOn )
        {  pq->lparen( _lparenOn ); _lparenOn = 0; }

    if ( _rparenOn )
        {  pq->rparen( _rparenOn ); _rparenOn = 0; }

    _current_op.push( pq );
}
```

Kun oikeanpuoleinen, päättävä sulku nähdään, käynnistetään evalRParen()-operaatio. Jos aktiivisen vasemmanpuoleisen sulun numero on suurempi kuin elementtien lukumäärä _current_op:ssa, niin silloin operaatio ei tee mitään. Muussa tapauksessa se suorittaa seuraavat vaiheet: se vetää _query_stack-pinosta ja hakee nykyisen, sijoittamattoman operandin. Se vetää _current_op-pinosta ja hakee nykyisen, keskeneräisen operaattorin. Sitten se käynnistää Query:n virtuaalisen add_op()-jäsenfunktion ja välittää keskeneräiselle operaattorille sijoittamattoman operandin. Lopuksi se työntää nyt valmiin operaattorin _query_stack-pinoon:

```
inline void
UserQuery::
evalRParen()
{
    if ( _paren < _current_op.size() )
    {
        Query *poperand = _query_stack.top();
        _query_stack.pop();

            Query *pop = _current_op.top();
            _current_op.pop();

            pop->add_op( poperand );
            _query_stack.push( pop );
    }
}
```

evalWord()-operaatio suorittaa seuraavat vaiheet: se etsii sanaa vastaavasta tekstitiedoston _word_map-sanojen sijaintikartasta. Ellei sanaa löydy, se hakee paikkavektorin ja varaa keosta uuden NameQuery-olion käynnistämällä kaksiparametrisen NameQuery-muodostajan. Jos elementtien lukumäärä _current_op:ssa on pienempi tai yhtä suuri kuin nähtyjen sulkujen lukumäärä, ei keskeneräistä operaattoria ole odottamassa NameQuery-operandia. Sen vuoksi NameQuery työnnetään _query_stack-pinoon; muussa tapauksessa haetaan keskeneräinen operaattori _current_op :sta. Query:n virtuaalinen add_op()-jäsenfunktio käynnistetään ja välitetään keskeneräiselle operaattorille NameQuery-olio ja juuri valmistunut operaattori työnnetään _query_stack-pinoon:

```
inline void
UserQuery::
evalWord( const string &query )
{
    NameQuery *pq;
    loc      *ploc;

    if ( ! _word_map->count( query ))
            pq = new NameQuery( query );
    else {
            ploc = ( *_word_map )[ query ];
            pq = new NameQuery( query, *ploc );
```

```
        }

        if ( _current_op.size() <= _paren )
                _query_stack.push( pq );
        else {
                Query *pop = _current_op.top();
                _current_op.pop();
                pop->add_op( pq );
                _query_stack.push( pop );
        }
}
```

Harjoitus 17.21

Tee UserQuery-luokalle tuhoaja, kopiointimuodostaja ja kopioinnin sijoitusoperaattori.

Harjoitus 17.22

Tee print()-funktio UserQuery-luokalle. Perustele, mitä valitsit sen tulostamiseen.

17.8 Kokoaminen

Tekstinkyselysovelluksemme main()-funktio näyttää seuraavalta:

```
#include "TextQuery.h"

int main()
{
    TextQuery tq;

    tq.build_text_map();
    tq.query_text();
}
```

build_text_map()-jäsenfunktion on nimetty uudelleen kohdan doit()-jäsenfunktioksi:

```
inline void
TextQuery::
build_text_map()
{
    retrieve_text();
    separate_words();
    filter_text();
    suffix_text();
    strip_caps();
    build_word_map();
}
```

query_text()-jäsenfunktio on kohdassa 6.14 määritellyn ilmentymän sijaisjäsenfunktio. Alkuperäisessä query_text()-toteutuksessamme sille uskottiin käyttäjältä kysely ja tulosten näyttö.

Olemme päättäneet säilyttää nuo velvollisuudet query_text()-jäsenfunktiossa ja toteuttaa toimenpiteet, jotka se suorittaa:

```
void
TextQuery::query_text()
{
    /* paikalliset oliot:
     *
     * text: sisältää kyselyn jokaisen sanan vuorollaan
     * query_text: vektori, joka sisältää käyttäjän kyselyn
     * caps: suodatin, joka kääntää isot kirjaimet pieniksi
     *
     * user_query : UserQuery-olio, johon kapseloidaan
                    käyttäjän kyselyn todellinen ratkaisu
     */
    string text;
    string caps( "ABCDEFGHIJKLMNOPQRSTUVWXYZ" );
    vector<string, allocator> query_text;
    UserQuery user_query;

    // alusta UserQuery:n  staattiset tietojäsenet
    NotQuery::all_locs( text_locations->second );
    AndQuery::max_col( &line_cnt );
    UserQuery::word_map( word_map );

    do {
        // poista aikaisempi kysely, jos sellainen on
        query_text.clear();

        cout << "Enter a query — please separate each item "
             << "by a space.\n"
             << "Terminate query (or session) "
             << "with a dot( . ).\n\n"
             << "==> ";

        /*
         * pyydä kysely vakiosyötöstä,
         * poista kaikki isot kirjaimet, sitten
         * työnnä se query_text:iin …
         *
         * huomaa: pitäisi tehdä kaikki käyttäjän kyselyn käsittely,
         *       jonka teemme itse tekstille ...
         */
        while( cin  >> text )
        {
            if ( text == "." )
                break;

            string::size_type pos = 0;
            while (( pos = text.find_first_of( caps, pos ))
```

```
                                        != string::npos )
                                    text[pos] = tolower( text[pos] );

                        query_text.push_back( text );
                }

                // ok: jos meillä on kysely, käsittele se …
                if ( ! query_text.empty() )
                {
                        // välitä kysely UserQuery-oliolle
                        user_query.query( &query_text );

                        // arvioi UserQuery
                        // palauta Query*-hierarkia …
                        // kohdassa 17.7 kuvataan tämä

                        // kysely on TextQuery:n Query*-jäsen
                        query = user_query.eval_query();

                        // ratkaise Query-hierarkia
                        // katso toteutus kohdasta 17.5
                        query->eval();

                        // ok: näytä ratkaisu
                        // TextQuery-jäsenfunktio
                        display_solution();

                        // saadaan aikaan ylimääräinen rivi käyttäjän päätteelle
                        cout << endl;
                }
        }
        while ( ! query_text.empty() ); cout << "Ok, bye!\n";
}
```

Ohjelman koko teksti on saatavilla Addison-Wesleyn ftp-paikasta, joka mainitaan kirjan takakannessa. Ohjelmamme viimeisenä kokeiluna käytimme sitä lukuisiin online-teksteihin. Ensimmäinen kysely tehtiin Herman Melvillen novellista *Bartleby*. Se kuvaa yhdistettyä AndQuery-kyselyä, jossa vierekkäiset sanat löytyvät riveiltä. (Huomaa, että sanat, jotka on laitettu vinoviivojen sisään, on tarkoitettu kursivoitaviksi.)

```
Enter a query — please separate each item by a space.
Terminate query (or session) with a dot( . ).
==> John && Jacob && Astor

    john ( 3 ) lines match
    jacob ( 3 ) lines match
    john && jacob ( 3 ) lines match
    astor ( 3 ) lines match
    john && jacob && astor ( 5 ) lines match
```

```
Requested query: john && jacob && astor
( 34 ) All who know me consider me an eminently /safe/ man. The late John Jacob
( 35 ) Astor, a personage little given to poetic enthusiasm, had no hesitation in
( 38 ) my profession by the late John Jacob Astor, a name which, I admit, I love to
( 40 ) bullion. I will freely add that I was not insensible to the late John Jacob
( 41 ) Astor's good opinion.
```

Tämä seuraava kysely, jossa kokeiltiin sulkuja ja yhdistettyjä operaattoreita, tehtiin Joseph Conradin novellille *Heart of Darkness*.

```
==> horror || ( absurd && mystery ) || ( North && Pole )

    horror ( 5 ) lines match
    absurd ( 8 ) lines match
    mystery ( 12 ) lines match
    ( absurd && mystery )  ( 1 ) lines match
    horror ||  ( absurd && mystery )  ( 6 ) lines match
    north ( 2 ) lines match
    pole ( 7 ) lines match
    ( north && pole )  ( 1 ) lines match
    horror ||  ( absurd && mystery )  ||  ( north && pole )
        ( 7 ) lines match

Requested query: horror ||  ( absurd && mystery )  ||  ( north && pole )
( 257 ) up I will go there.' The North Pole was one of these
( 952 ) horrors. The heavy pole had skinned his poor nose.
( 3055 ) some lightless region of subtle horrors, where pure,
( 3673 ) " 'The horror! The horror!'
( 3913 ) the whispered cry, 'The horror! The horror! '
( 3957 ) absurd mysteries not fit for a human being to behold.
( 4088 ) wind. 'The horror! The horror!'
```

Viimeinen kysely tehtiin katkelmaan Henry Jamesin tarinasta *Portrait of a Lady*. Se kuvaa yhdistettyä kyselyä ja kyseessä on suuri tekstitiedosto.

```
==> clever && trick || devious

    clever ( 46 ) lines match
    trick ( 12 ) lines match
    clever && trick ( 2 ) lines match
    devious ( 1 ) lines match
    clever && trick || devious ( 3 ) lines match

Requested query: clever && trick || devious
( 13914 ) clever trick she had guessed. Isabel, as she herself grew older,
( 13935 ) lost the desire to know this lady's clever trick. If she had
( 14974 ) desultory, so devious, so much the reverse of processional. There were
```

Harjoitus 17.23

Käyttäjän kyselyn käsittelymme epäonnistuu nykyisellään siinä, että se ei käytä samaa esikäsittelyä jokaiselle sanalle kuten alkupäässä tehdään tekstiä muodostettaessa; katso kohdat 6.9 ja 6.10. Siten esimerkiksi käyttäjä, joka haluaa löytää sanan maps, huomaa, että vain map löydetään tekstistä. Muokkaa query_text() niin, että se tekee samanlaisen esikäsittelyn.

Harjoitus 17.24

Kyselyjärjestelmää voitaisiin kehittää edelleen toisella InclusiveAndQuery-kyselyllä; ehkäpä esittäen yhdellä &-merkillä. Se saisi totuusarvon tosi, jos molemmat sanat olisivat samalla rivillä sen sijaan, että ne olisivat vierekkäin. Esimerkiksi rivillä

```
We were her pride of ten, she named us
```

saa InclusiveAndQuery

```
pride & ten
```

arvon tosi, kun taas alkuperäinen AndQuery

```
pride && ten
```

saa arvon epätosi. Tee tarvittava tuki InclusiveAndQuery-kyselylle.

Harjoitus 17.25

Nykyinen toteutuksemme display_solution()-funktiosta, joka on seuraavassa, tulostaa vain vakiotulostukseen. Järkevämpi toteutus olisi mahdollistaa, että käyttäjä voi ilmaista ostream-virran, jonne tulostus ohjataan. Muokkaa display_solution()-funktiota niin, että se mahdollistaa käyttäjäkohtaisen ostream-olion. Mitkä muut muutokset ovat tarpeen UserQuery-luokkamäärittelyssä?

```
void TextQuery::
display_solution()
{
    cout << "\n"
       << "Requested query: "
       << *query << "\n\n";

    const set<short> *solution = query->solution();
    if ( ! solution->size() ) {
        cout << "\n\tSorry, "
                << " no matching lines were found in text.\n"
                << endl;
        return;
    }

    set<short>::const_iterator
        it = solution->begin(),
        end_it = solution->end();
```

```
    for ( ; it != end_it; ++it ) {
        int line = *it;

        // älä hämmennä käyttäjää tekstiriveillä, jotka alkavat arvosta 0 ...
        cout << "( " << line+1 << " ) "
             << (*lines_of_text)[line] << '\n';
    }

    cout << endl;
}
```

Harjoitus 17.26

Se, mitä TextQuery-luokkamme todella tarvitsee, on kyky hyväksyä komentorivin argumentteja käyttäjältä.

(a) Muotoile mahdollinen komentorivisyntaksi tekstinkyselyjärjestelmällemme.

(b) Osoita lisätietojäsenet ja -jäsenfunktiot, jotka ovat tarpeen.

(c) Luonnostele komentorivipiirteen toteutus (katso kohdasta 7.8 esimerkkiä).

Harjoitus 17.27

Mieti mahdollisena ohjelmointiprojektina jotain seuraavista parannuksista kyselyjärjestelmäämme:

(a) Esittele tuki, jossa AndQuery esitellään string-oliona kuten “Motion Picture Screen Cartoonists”.

(b) Esittele tuki, jossa sanoja vertaillaan sen perusteella, ovatko ne samassa lauseessa sen sijaan, että ne olisivat samalla rivillä.

(c) Esittele historiajärjestelmä, jossa käyttäjä voi viitata aikaisempaan kyselyyn numerolla, ehkä lisäten sen siihen tai yhdistäen sen toiseen kyselyyn.

(d) Sen sijaan, että näytetään osumien lukumäärä ja kaikki täsmäävät rivit, mahdollista, että käyttäjä voi ilmaista näytettävien rivien alueen sekä välikyselylle että lopulliselle kyselylle. Esimerkiksi:

```
==> John && Jacob && Astor

(1)     john ( 3 ) lines match
(2)     jacob ( 3 ) lines match
(3)     john && jacob ( 3 ) lines match
(4)     astor ( 3 ) lines match
(5)     john && jacob && astor ( 5 ) lines match

// Uusi piirre: anna käyttäjän valita, mikä kysely näytetään
// Käyttäjä kirjoittaa numeron
==> display?  3
```

```
// Sitten järjestelmä kysyy, kuinka monta riviä näytetään
// Enter-näppäimen painallus näyttää kaikki tai käyttäjä voi kirjoittaa yhden rivinumeron tai alueen
==> how many( return displays all, else enter single line or range) 1──3
```