16

Luokkamallit

Tässä luvussa kuvataan luokkamallit ja kuinka niitä määritellään. Luokkamalli on ohje sellaisen luokan luomiselle, jonka yksi tai useampi tyyppi on parametroitu. Aloittelevana C++-ohjelmoijana on mahdollista käyttää luokkamalleja ymmärtämättä mekanismia mallimäärittelyiden ja instantiointien takana. Itse asiassa olemme käyttäneet C++-vakiokirjastoon luotuja luokkamalleja (kuten vektori, lista jne.) kautta tämän kirjan ilman tarvetta kuvata mallimekanismia tarkemmin. Vain edistyneemmät C++-ohjelmoijat määrittelevät omat luokkamallinsa ja käyttävät tässä luvussa kuvattuja mekanismeja. Tämän luvun materiaali on siitä syystä tutustumisjakso C++:ssa pitemmälle menevään aiheeseen.

Luku on jaettu tutustumis- ja jatko-osiin. Tutustumisjaksoissa näytetään, kuinka luokkamalleja määritellään, kuvataan luokkamallien yksinkertaisia käyttötilanteita ja sitä, kuinka luokkamalleja instantioidaan. Tutustumisjaksossa katsotaan myös erilaisia jäseniä, joita voidaan määritellä luokkamalliin: jäsenfunktiot, staattiset tietojäsenet ja sisäkkäiset tyypit. Jatko-osassa esitellään materiaalia, joka on tarpeellista tuotantotasoisten sovellusten kirjoittamiseen. Katsomme ensin sitä, kuinka kääntäjä instantioi malleja, ja vaatimuksia, joita tämä asettaa ohjelmiemme järjestykselle. Sitten esittelemme, kuinka määrittelemme erikoistamisia ja osittaiserikoistamisia luokkamalliin tai luokkamallin jäseneen. Sitten luvussa katsotaan kahta aihetta, jotka kiinnostavat luokkamallien suunnittelijoita: kuinka luokkamallien määrittelyiden nimet ratkaistaan ja kuinka luokkamalleja voidaan määritellä nimiavaruuksiin. Luku päättyy laajempaan esimerkkiin luokkamallin määrittelystä ja käytöstä.

16.1 Luokkamallin määritteleminen

Oletetaan, että haluamme määritellä luokan, joka tukee jonomekanismia. Jono on tietorakenne oliokokoelmalle, jossa oliot lisätään yhteen päähän eli taakse ja poistetaan toisesta päästä eli edestä. Jonon käyttäytymistä kuvaa sanonta “ensiksi sisään, ensiksi ulos” eli *FIFO* (*first in*

first out). (C++-vakiokirjastossa on määritelty jonotyyppi (queue), ja se on kuvattu lyhyesti kohdassa 6.17. Tässä luvussa määrittelemme oman yksinkertaisen jonotyypin esitelläksemme luokkamalleja.)

Päätämme, että Queue-luokkamme tukee seuraavia operaatioita:

- Lisää jäsenen jonon taakse:

```
void add( item );
```

- Poistaa jäsenen jonon edestä:

```
item remove();
```

- Päättelee, onko jono tyhjä:

```
bool is_empty();
```

- Päättelee, onko jono täysi:

```
bool is_full();
```

Queue-luokkamme määrittely voisi näyttää tältä:

```
class Queue {
public:
    Queue();
    ~Queue();

    Type& remove();
    void add( const Type & );
    bool is_empty();
    bool is_full();
private:
    // ....
};
```

Kysymys kuuluu, mitä tyyppiä tulisi käyttää Type:n tilalla? Olettakaamme, että toteutamme Queue-luokan korvaamalla Type:n int-tyypillä. Queue-luokka on siten määritelty käsittelemään int-tyyppisiä oliokokoelmia. Jos ohjelmoija sijoittaisi jonkin muun tyypin näihin olioihin, se joko konvertoitaisiin int-tyypiksi tai ellei konversiota olisi olemassa, sijoitus saisi aikaan käännösvirheen. Esimerkiksi:

```
Queue qObj;
string str( "vivisection" );

qObj.add( 3.14159 ); // ok: jonoon lisätty jäsen on == 3
qObj.add( str );     // virhe: konversiota ei ole string-tyypistä int-tyyppiin
```

Koska jokainen kokoelman olio on int-tyyppi, C++:n tyyppijärjestelmä takaa, että vain int-tyyppiset tai sellaiset arvot, jotka voidaan konvertoida int-tyyppisiksi, voidaan sijoittaa Queue-tyyppisiin olioihin. Tämä on hyvä asia, kun ohjelmoija haluaa käyttää jonoa, joka muodostuu int-tyyppisistä olioista. Tämä ei kuitenkaan ole silloin hyvä asia, kun ohjelmoija haluaa käyttää Queue-luokkaa double-tyypille, char-tyypille, kompleksiluvulle tai merkkijonolle.

Eräs menetelmä on käyttää raakaa voimaa. Ohjelmoija kopioi Queue-luokan toteutuksen, muokkaa sitä niin, että se toimii double-tyypeillä, sitten kompleksiluvulla, sitten merkkijonoilla jne. Ja koska luokkanimiä ei voi ylikuormittaa, pitää jokaiselle toteutukselle antaa yksilöllinen nimi: IntQueue, DoubleQueue, ComplexQueue, StringQueue. Kun uutta luokkatyyppiä tarvitaan, kopioidaan koodi, muutetaan ja nimetään se uudelleen.

Mitä ongelmia on tässä luokkatyypin kopiointimenetelmässä? On kirjaimellinen ongelma, joka koskee yksilöllisesti nimettäviä Queue-luokkia. On myös hallinnollinen monimutkaisuus — kuvittele, että aiot levittää IntQueue-luokan yleisiin osiin muutoksen jokaiseen ilmentymään. Yleensä käsin tehty kopiointi yksittäisistä tyypeistä on loputon prosessi ja vaikea hallita.

Onneksi C++:n mallipiirre tarjoaa ratkaisun luokkatyyppien automaattiselle generoinnille. Voimme käyttää luokkamallia ja generoida Queue-luokan automaattisesti mille tahansa tyypille. Queue-luokan luokkamalli voisi näyttää tältä:

```
template <class Type>
class Queue {
public:
    Queue();
    ~Queue();

    Type& remove();
    void add( const Type & );
    bool is_empty();
    bool is_full();
private:
    // ....
};
```

Ohjelmoija kirjoittaa

```
Queue<int> qi;
Queue< complex<double> > qc;
Queue<string> qs;
```

kun haluaa generoida Queue-luokan vuorostaan int-tyypeistä, kompleksiluvuista ja string-tyypeistä.

Queue-luokkamme toteutus on esitetty seuraavissa kohdissa ja kuvaa luokkamallien määrittelyä ja käyttöä. Toteutuksessa käytetään luokkamalliabstraktioparia:

1. Itse Queue-luokkamallissa on aikaisemmin kuvattu julkinen rajapinta ja tietojäsenpari front ja back. Queue-luokkamalli on toteutettu linkitettynä listana.
2. QueueItem-luokkamalli edustaa solmua Queue:n linkitetyssä listassa. Jokainen jonoon tuleva jäsen tallennetaan QueueItem-olioon. QueueItem-olio sisältää tietojäsenparin value ja next. value-tyypin todellinen tyyppi vaihtelee Queue-ilmentymästä toiseen. next on linkki seuraavaan QueueItem-olioon jonossa.
3.

Ennen kuin katsomme näiden mallien toteutuksia tarkemmin, tutkikaamme edelleen, kuinka malleja esitellään ja määritellään. Tässä on QueueItem-luokkamallin esittely:

```
template <class T>
    class QueueItem;
```

Avainsana template aloittaa aina luokkamallin sekä määrittelyn että esittelyn. Tämän avainsanan jälkeen tulee pilkuin eroteltu luettelo malliparametreista kulmasulkujen sisällä (< >). Tätä listaa sanotaan luokkamallin *malliparametriluetteloksi*. Se ei voi olla tyhjä. Malliparametri voi olla tyyppiparametri tai ei mikään tyyppi, jota edustaa vakiolauseke.

Mallin *tyyppiparametri* muodostuu avainsanasta class tai typename ja tunnuksesta. Malliparametriluettelossa avainsanoilla class ja typename on sama merkitys. (Avainsanaa typename ei tueta C++-esistandardin toteutuksissa. Kohdassa 10.1 tuodaan esille tarkemmin, miksi tämä avainsana lisättiin C++:aan: koska sitä täytyy joskus käyttää kääntäjän ohjaamiseen mallimäärittelyiden tulkinnassa.) Nämä avainsana ilmaisevat, että parametrinimi, joka tulee sen jälkeen, edustaa sisäistä tai käyttäjän määrittelemää tyyppiä. Esimerkiksi aiemmin käsitellyn QueueItem-luokkamallin esittelyssä on yksi mallityyppiparametri nimeltään T. Mikä tahansa sisäinen tai käyttäjän määrittelemä tyyppi kuten int , double , char* , complex tai string kelpaa T:n argumentiksi:

Luokkamallilla voi olla useita tyyppiparametreja:

```
template <class T1, class T2, class T3>
    class Container;
```

Kuitenkin jokaisen mallityyppiparametrin eteen pitää laittaa avainsana class tai typename. Esimerkiksi seuraava malliesittely on virheellinen:

```
// virhe: pitää olla <typename T, class U> tai
//              <typename T, typename U>
template <typename T, U>
    class collection;
```

Kun tyyppiparametri on esitelty, toimii se tyyppimääreenä luokkamallimäärittelyn loppuun saakka. Sitä voidaan käyttää luokkamallimäärittelyssä täsmälleen samalla lailla kuin sisäistä tai käyttäjän määrittelemää tyyppiä voidaan käyttää mallittoman luokan määrittelyssä. Tyyppiparametria voidaan käyttää esiteltäessä esimerkiksi tietojäseniä, jäsenfunktioita, sisäkkäisten luokkien jäseniä jne.

Mallin *tyypitön parametri* muodostuu tavallisesta parametriesittelystä. Tyypitön parametri ilmaisee, että parametrin nimi edustaa mahdollista arvoa. Tämä arvo edustaa vakiota luokkamallimäärittelyssä. Esimerkiksi Buffer-luokkamallissa voi olla tyyppiparametri, joka ilmaisee sisältämiensä elementtien tyypin ja tyypitön parametri, joka on vakioarvo ja edustaa sen kokoa. Esimerkiksi:

```
template <class Type, int size>
    class Buffer;
```

Malliparametriluettelon jälkeen tulee luokan määrittely tai esittely. Paitsi, että malliparametrit ovat mukana, luokkamallin määrittely näyttää samalta kuin mallittoman luokan määrittely:

```
template <class Type>
class QueueItem {
public:
    // ...
private:
    // Type edustaa tietojäsenen tyyppiä
    Type item;
    QueueItem *next;
};
```

Esimerkissä on käytetty Type:ä ilmaisemaan item-tietojäsenen tyyppi. Ohjelman aikana Type tullaan korvaamaan monilla sisäisillä ja käyttäjän määrittelemillä tyypeillä. Tätä tyyppien korvaamista kutsutaan *mallin instantioinniksi* (*template instantiation*).

Malliparametrin nimeä voidaan käyttää sen jälkeen, kun se on esitelty malliparametriksi aina mallin esittelyn tai määrittelyn loppuun saakka. Jos globaalilla viittausalueella on esitelty muuttuja, joka on samanniminen kuin malliparametri, tuo nimi jää piiloon. Seuraavassa esimerkissä item ei ole double-tyyppinen. Sen tyyppi on malliparametrin tyyppi:

```
typedef double Type;

template <class Type>
class QueueItem {
public:
    // ...
private:
    // item ei ole double-tyyppinen
    Type item;
    QueueItem *next;
};
```

Malliparametrin nimeä ei voi käyttää luokan jäsenen nimenä, joka on esitelty luokkamallin määrittelyssä:

```
template <class Type>
class QueueItem {
public:
    // ...
private:
    // virhe: jäsenellä ei voi olla samaa nimeä kuin
    //        malliparametrilla Type
    typedef double Type;
    Type item;
    QueueItem *next;
};
```

Malliparametrin nimi voidaan mainita vain kerran malliparametriluettelossa. Esimerkiksi seuraava saa aikaan virheen käännöksen aikana:

```
// virhe: Type-nimisen malliparametrin uudelleenkäyttö ei kelpaa
template <class Type, class Type>
    class container;
```

Malliparametrin nimeä voidaan käyttää uudelleen muiden luokkamallien esittelyissä ja määrittelyissä:

```
// ok: 'Type'-nimen käyttö muissa malleissa
template <class Type>
    class QueueItem;

template <class Type>
    class Queue;
```

Malliparametrien nimien ei tarvitse olla samoja luokkamallin muissa jatkoesittelyissä ja määrittelyissä. Seuraavat kolme QueueItem-esittelyä viittaavat kaikki samaan luokkamalliin:

```
// kaikki kolme QueueItem-esittelyä
// viittaavat samaan luokkamalliin

// mallin esittelyt
template <class T> class QueueItem;
template <class U> class QueueItem;

// mallin varsinainen määrittely
template <class Type>
    class QueueItem { ... };
```

Luokkamallin parametreilla voi olla oletusargumentit. Tämä pätee, oli parametri sitten tyyppiparametri tai tyypitön parametri. Aivan kuten funktion parametrien oletusargumenttien yhteydessä (esiteltiin kohdassa 7.3), malliparametrin oletusargumentti on tyyppi tai arvo, jota käytetään, ellei argumenttia ole määritetty mallia instantioitaessa. Oletusargumentin tulisi olla tyyppiä, joka sopii valtaosaan luokkamallin instantioinneista. Jos seuraavassa esimerkissä ei mallin instantioinnin yhteydessä määritetä kokoa Buffer-oliolle sen nimen jälkeen, on instantioidun Buffer-olion koko 1024.

```
template <class Type, int size = 1024>
    class Buffer;
```

Luokkamallin jatkoesittelyissä voidaan antaa lisää oletusargumentteja malliparametreille. Aivan kuten funktion parametrien oletusargumenttien yhteydessä, oikeanpuoleisimmalle alustamattomalle parametrille pitää antaa oletusargumentti ennen kuin yhtään oletusargumenttia sen vasemmalla puolella oleville parametreille voidaan antaa. Esimerkiksi:

```
template <class Type, int size = 1024>
    class Buffer;
```

```
// ok: otetaan huomioon kahden esittelyn oletusargumentit
template <class Type = string , int size>
    class Buffer;
```

(Huomaa, että malliparametrien oletusargumentteja ei tueta C++-esistandardin toteutuksissa. Monet tämän kirjan esimerkit, kuten esimerkiksi luvussa 12, kirjoitettiin ilman oletusargumentteja, jotta ne kääntyisivät C++-esistandardin mukaisessa toteutuksessa.)

Luokkamallin määrittelyn sisällä voidaan käyttää luokkanimeä tyyppimääreenä aina siellä, missä mallitontakin luokkanimeä voidaan käyttää. Tässä on esimerkiksi täydellisempi versio QueueItem-luokkamallin määrittelystä:

```
template <class Type>
class QueueItem {
public:
    QueueItem( const Type & );
private:
    Type item;
    QueueItem *next;
};
```

Huomaa, että jokainen QueueItem-luokkamallin nimen esiintymä luokkamallin määrittelyssä on lyhenne merkinnästä

```
QueueItem<Type>
```

Tätä lyhennettyä merkintätapaa voidaan käyttää vain itse QueueItem-luokkamallin määrittelyssä (ja sen jäsenten määrittelyissä, jotka esiintyvät luokkamallin määrittelyn ulkopuolella kuten tulemme näkemään seuraavissa kohdissa). Kun QueueItem-luokkamallia on käytetty tyyppimääreenä toisessa mallimäärittelyssä, pitää määrittää koko malliparametriluettelo. Seuraavassa esimerkissä luokkamallia on käytetty display-luokkamallin määrittelyssä. Tässä tapauksessa QueueItem-luokkamallin nimen jälkeen pitää tulla malliparametrit, kuten tapauksessa QueueItem<Type>.

```
template <class Type>
void display( QueueItem<Type> &qi )
{
    QueueItem<Type> *pqi = &qi;
    // ...
}
```

16.1.1 Queue- ja QueueItem-luokkamallien määrittelyt

Tässä on Queue-luokkamallin määrittely. Se sijoitetaan otsikkotiedostoon nimeltään Queue.h yhdessä QueueItem-luokkamallin määrittelyn kanssa.

```
#ifndef QUEUE_H
#define QUEUE_H

// QueueItem-luokkamallin määrittely
template <class T> class QueueItem;
```

```
template <class Type>
class Queue {
public:
    Queue() : front( 0 ), back ( 0 ) { }
    ~Queue();

    Type& remove();
    void add( const Type & );
    bool is_empty() const {
        return front == 0;
    }
private:
    QueueItem<Type> *front;
    QueueItem<Type> *back;
};
#endif
```

Queue-luokkamallin määrittelyssä voidaan Queue-nimen käytön yhteydessä jättää pois parametriluettelo <Type>. Parametriluetteloa ei voi kuitenkaan jättää pois, kun Queue-määrittely viittaa QueueItem-luokkamalliin. Esimerkiksi seuraava front-esittely on virheellinen:

```
template <class Type>
class Queue {
public:
    // ...
private:
    // virhe: QueueItem ei ole tunnettu tyyppi
    QueueItem *front;
};
```

Harjoitus 16.1

Yksilöi, mitkä seuraavista luokkamallien esittelyistä (tai esittelypareista) ovat kelpaamattomia (vai ovatko mitkään?).

(a)
```
template <class Type>
    class Container1;

template <class Type, int size>
    class Container1;
```

(b)
```
template <class T, U, class V>
    class Container2;
```

(c)
```
template <class C1, typename C2>
    class Container3 {};
```

(d)
```
template <typename myT, class myT>
    class Container4 {};
```

```
(e)  template <class Type, int *ptr>
         class Container5;

     template <class T, int *pi>
         class Container5;

(f) template <class Type, int val = 0>
         class Container6;
     template <class T = complex<double>, int v>
         class Container6;
```

Harjoitus 16.2

Seuraava List-määrittely on virheellinen. Kuinka korjaisit sen?

```
template <class elemenType>
class ListItem;

template <class elemType>
class List {
public:
    List<elemType>()
      : _at_front( 0 ), _at_end( 0 ), _current( 0 ), _size( 0 )
      {}
    List<elemType>( const List<elemType> & );
    List<elemType>& operator=( const List<elemType> & );

    ~List();

    void insert( ListItem *ptr, elemType value );
    int  remove( elemType value );

    ListItem *find( elemType value );

    void display( ostream &os = cout );
    int size() { return _size; }
private:
    ListItem *_at_front;
    ListItem *_at_end;
    ListItem *_current;
    int _size;
};
```

16.2 Luokkamallin instantiointi

Luokkamallin määrittelyssä määrätään, kuinka yksilöllisiä luokkia voidaan muodostaa annetulla yhdellä tai useammalla todellisella tyypillä tai arvolla. Queue-luokkamallin määrittely toimii mallina Queue-luokkien tyyppikohtaisten ilmentymien generoinnille. Esimerkiksi int-tyyppisten olioiden Queue-luokka luodaan automaattisesti geneerisestä luokkamallin määrittelystä, kun ohjelmoija kirjoittaa

```
Queue<int> qi;
```

Tätä luokan generointia geneerisestä luokkamallin määrittelystä kutsutaan *mallin instantioinniksi* (*template instantiation*). Kun int-tyyppisten olioiden Queue-luokka instantioidaan, jokainen Type-malliparametrin esiintymä luokkamallin määrittelyssä korvataan int-tyypillä. Queue-luokasta tulee kirjaimellisesti

```
class Queue<int> {
public:
    Queue<int>() : front( 0 ), back ( 0 ) { }
    ~Queue<int>();

    int& remove();
    void add( const int & );
    bool is_empty() const {
        return front == 0;
    }
private:
    QueueItem<int> *front;
    QueueItem<int> *back;
};
```

Samalla tavalla kuin Queue-luokka luodaan string-tyyppisille olioille, ohjelmoija kirjoittaa

```
Queue<string> qs;
```

Tässä tapauksessa jokainen Type-malliparametrin esiintymä luokkamallin määrittelyssä korvataan string-tyypillä. Oliot qi ja qs ovat luokkatyyppisiä olioita.

Erityyppisten luokkamallien instantioinneilla ei ole mitään erityistä suhdetta toisiinsa. Sen sijaan jokainen luokkamallin instantiointi muodostaa riippumattoman luokkatyypin. Esimerkiksi int-tyypisellä Queue-instantioinnilla ei ole mitään pääsyä string-tyypin Queue-instantioinnin ei-julkisiin jäseniin.

Luokkamallin instantioinnin nimi on Queue<int> tai Queue<string>. Sanasia <int> ja <string>, jotka tulevat luokkamallin Queue-nimen jälkeen, sanotaan malliargumenteiksi. Malliargumentit pitää määrittää pilkuin erotetulla luettelolla kulmasulkujen sisällä (< >). Luokkamallin instantioinnin pitää aina määrittää malliargumentit eksplisiittisesti. Toisin kuin funktiomallin instantioinnin malliargumentteja, luokkamallin instantioinnin malliargumentteja ei päätellä koskaan siitä yhteydestä, jossa niitä on käytetty luokkamallin instantioinnissa:

```
Queue qs;  // virhe: minkä mallin instantiointi?
```

Yleinen ohjelma voi käyttää Queue-luokkamallin instantiointia siellä, missä mallitontakin luokkatyyppiä voidaan käyttää:

```
// paluutyyppi ja kaksi parametria ovat Queue:n instantiointeja
extern Queue< complex<double> >
    foo( Queue< complex<double> > &, Queue< complex<double> > & );

// osoitin Queue-instantioinnin jäsenfunktioon
bool (Queue<double>::*pmf)() = 0;

// 0:n eksplisiittinen tyyppimuunnos osoittimeksi Queue:n instantiointiin
Queue<char*> *pqc = static_cast< Queue<char*>* > ( 0 );
```

Luokkatyyppisiä olioita, jotka ovat Queue-luokkamallin instantiointeja, esitellään ja käytetään täsmälleen samalla tavalla kuin mallittomia luokkatyyppejä:

```
extern Queue<double> eqd;
Queue<int> *pqi = new Queue<int>;
Queue<int> aqi[1024];

int main() {
    int ix;
    if ( ! pqi->is_empty() )
        ix = pqi->remove();
    // ...
    for ( ix = 0; ix < 1024; ++ix )
        eqd[ ix ].add( ix );
    // ...
}
```

Mallin esittely tai määrittely voi viitata luokkamalliin tai luokkamallin instantiointiin:

```
// funktiomallin esittely
template <class Type>
void bar( Queue<Type> &,  // viittaa geneeriseen malliin
      Queue<double> & // eikä mallin instantiointiin
)
```

Kuitenkin mallimäärittelyn asiayhteyden ulkopuolella voidaan käyttää vain luokkamallin instantiointeja. Esimerkiksi mallittoman funktion pitää aina määrittää, mitä tiettyä Queue-luokkamallin instantiointia se käyttää:

```
void foo( Queue<int> &qi )
{
    Queue<int> *pqi = &qi;
    // ...
}
```

Luokkamalli instantioidaan vain, kun instantionnin nimeä käytetään yhteydessä, joka vaatii luokkamäärittelyn olemassaoloa. Eivät kaikki luokan käyttötilanteet vaadi sitä, että luokkamäärittely tunnetaan. Ei ole esimerkiksi tarpeen tietää luokan määrittelyä ennen kuin osoittimia ja viittauksia luokkaan voidaan esitellä. Esimerkiksi:

```
class Matrix;
Matrix *pm; // ok: Matrix-luokan määrittelyä ei tarvita

void inverse( Matrix & ); // myös ok
```

Tästä syystä, kun esitellään osoittimia ja viittauksia luokkamallin instantiointiin, se ei saa aikaan luokkamallin instantiointia. (Tässä pitää mainita, että jotkut C++-esistandardia tukevat toteutukset instantioivat mallin, kun instantioinnin nimi mainitaan ohjelmatekstissä ensimmäisen kerran.) Esimerkiksi seuraava foo()-funktio esittelee osoittimen ja viittauksen luokkamallin instantiointiin Queue<int>. Nämä esittelyt eivät kuitenkaan aiheuta Queue-mallin instantiointia:

```
// Queue<int> jää instantioimatta sen käyttötilanteissa foo()-funktiossa
void foo( Queue<int> &qi )
{
    Queue<int> *pqi = &qi;
    // ...
}
```

Luokkamäärittely tarvitaan, kun tämän luokkatyyppinen olio määritellään. Esimerkiksi seuraavan esimerkin obj1-olion määrittely on virheellinen. Tämä oliomäärittely vaatii, että Matrix:in koko pitää olla kääntäjän tiedossa, jotta se voisi varata oikean määrän muistia obj1-oliolle:

```
class Matrix;
Matrix obj1; // virhe: Matrix-luokkaa ei ole määritelty

class Matrix { ... };
Matrix obj2; // ok
```

Siitä syystä luokkamalli instantioidaan, kun olio määritellään tyypiksi, joka on luokkamallin instantiointi. Seuraavassa esimerkissä qi-olion määrittely saa aikaan Queue<int>-mallin instantioinnin:

```
Queue<int> qi; // Queue<int> instantioidaan
```

Queue<int>-luokan määrittely tulee tunnetuksi kääntäjälle tässä kohdassa, jota sanotaan Queue<int>-luokan *instantiointikohdaksi*.

Samalla tavalla, jos osoitin tai viittaus viittaa luokkamallin instantiointiin, niin luokkamalli instantioidaan vain silloin, kun oliota, johon osoitin tai viittaus viittaa, tutkitaan. Aikaiemmin määritellyn foo()-funktion Queue<int> instantioidaan, jos pqi-osoitinta käytetään käänteisesti, jos qi-osoitinta käytetään sen olion arvon saamiseksi, johon se viittaa, tai jos pqi- tai qi-osoitinta käytetään Queue<int>-tietojäsenten tai -jäsenfunktioiden käsittelyyn:

```
void foo( Queue<int> &qi )
{
    Queue<int> *pqi = &qi;

    // Queue<int> instantioidaan jäsenfunktion kutsun vuoksi
    pqi->add( 255 );
    // ...
}
```

Queue<int>-luokan määrittelystä tulee tunnettu kääntäjälle, ennen kuin sen jäsenfunktiota add() kutsutaan foo()-funktiossa.

Muista, että Queue-luokkamallin määrittely viittaa QueueItem-luokkamalliin seuraavasti:

```
template <class Type>
class Queue {
public:
    // ...
private:
    QueueItem<Type> *front;
    QueueItem<Type> *back;
};
```

Kun Queue instantioidaan int-tyyppisenä, ovat Queue<int>-instantioinnin front- ja back-jäsenet osoittimia QueueItem<int>-instantiointiin. Sen vuoksi Queue<int>-instantiointi viittaa int-tyyppisenä instantioituun QueueItem-luokkamalliin. Koska kuitenkin nämä jäsenet ovat osoittimia, QueueItem<int>-tyyppi instantioidaan vain, kun näitä jäseniä käytetään käänteisesti Queue<int>-luokan jäsenfunktioissa.

Olemme päättäneet, että QueueItem on tarkoituksellisesti apuluokka, joka auttaa Queue-luokan toteutusta. Ei ole tarkoitus, että pääohjelma käyttäisi sitä. Siten ohjelmat käsittelevät vain Queue-luokkaolioita. QueueItem-luokkamallin instantioinnit saavat aikaan vain Queue-luokkamallin ja sen jäsenten instantioinnit. Seuraavissa kohdissa katsomme luokkamallin jäsenten instantiointia.

Riippuen tyypeistä, joilla luokkamalli instantioidaan, pitää ottaa huomioon joitakin suunnittelunäkökohtia, kun luokkamallia määritellään. Huomaatko esimerkiksi, miksi seuraavaa QueueItem-muodostajan määrittelyä on todennäköisesti mahdoton hyväksyä suurimmalle osalle tyyppi-instantiointeja?

```
template <class Type>
class QueueItem {
public:
    QueueItem( Type ); // huono suunnittelupäätös
    // ...
};
```

Tämä QueueItem-muodostajan määrittely toteuttaa argumentin välityksen arvonaan. Tämä toimii riittävästi, kun QueueItem instantioidaan sisäisellä tyypillä (kuten esimerkiksi QueueItem<int>-instantioinnissa). Kuitenkin, kun QueueItem instantioidaan suurella luokkatyypillä (jollainen on esimerkiksi Matrix), siitä tulee suoritusajan kannalta kestämätön. (Kohdassa 7.3 käsitellään vaikutuksia suorituskykyyn, kun esitellään arvona välitettäviä parametreja vastakohtana, kun ne esitellään viittausparametreina.) Tästä syystä argumentti muodostajalle on esitelty viittauksena const-tyyppiin:

```
QueueItem( const Type & );
```

Toinen suunnittelunäkökohta liittyy tämän muodostajan toteutukseen. Seuraava muodostajamäärittely on hyväksyttävä, jos tyypillä, jolla QueueItem on instantioitu, ei ole vastaavaa muodostajaa:

```
template <class Type>
class QueueItem {
    // ...
public:
    // todennäköisesti tehoton
    QueueItem( const Type &t ) {
        item = t; next = 0;
    }
};
```

Jos malliargumentti on luokkatyyppi, jolla on muodostaja (esimerkkinä string), se johtaa siihen, että item alustetaan kahdesti! String-tyypille käynnistetään oletusmuodostaja, joka alustaa item:in ennen QueueItem-muodostajan rungon suoritusta. Juuri muodostettuun item:iin sijoitetaan sitten jäsenittäin. Jos item alustetaan eksplisiittisesti muodostajan jäsenen alustusluettelolla QueueItem-muodostajan määrittelyssä, ratkaistaan tämä ongelma:

```
template <class Type>
class QueueItem {
    // ...
public:
    // item alustetaan muodostajassa jäsenen alustusluettelolla
    QueueItem( const Type &t )
        : item(t) { next = 0; }
```

```
};
```

(Kohdassa 14.5 käsitellään jäsenen alustusluetteloita ja sitä, milloin sekä kuinka niitä tulisi käyttää.)

16.2.1 Malliargumentit tyypittömille parametreille

Luokan malliparametri voi olla tyypitön. On olemassa joitakin rajoituksia eräille malliargumenteille, joita voidaan käyttää tyypittömille malliparametreille. Tutkimme näitä rajoituksia tässä. Seuraavassa esimerkissä käytetään Screen-luokkaa, joka esiteltiin ensimmäisen kerran luvussa 13. Se on määritelty uudelleen tässä malliksi ja parametroitu korkeudellaan ja leveydellään:

```
template <int hi, int wid>
class Screen {
public:
    Screen() : _height( hi ), _width( wid ), _cursor ( 0 ),
               _screen( hi * wid, '#' )
             { }
    // ...
private:
    string          _screen;
    string::size_type _cursor;
    short           _height;
    short           _width;
};

typedef Screen<24,80> termScreen;
termScreen hp2621;

Screen<8,24> ancientScreen;
```

Lausekkeen, johon tyypitön parametri sidotaan, pitää olla vakiolauseke. Tämä tarkoittaa, että pitää olla mahdollista ratkaista se käännöksen aikana. Edellisessä esimerkissä typedef-nimi termScreen viittaa malli-instantiointiin Screen<24,80>. Parametrien hi ja wid malliargumentit ovat vastaavasti 24 ja 80. Molemmissa tapauksissa malliargumentti on vakiolauseke.

Kuitenkin, jos ByfPtr-luokkamalli on määritelty tässä, sen instantiointi johtaa käännösvirheeseen, koska osoittimen arvoa, joka saadaan tuloksena new()-operaattorin käynnistyksestä, ei tiedetä ennen kuin suorituksen aikana:

```
template <int *ptr> class BufPtr { ... };

// virhe: malliargumenttia ei voida ratkaista käännöksen aikana
BufPtr< new int[24] > bp;
```

Samalla tavalla ei const-tyypittömän olion arvo ole vakiolauseke. Sitä ei voi käyttää malliargumenttina tyypittömälle malliparametrille. Kuitenkin minkä tahansa nimiavaruuden viittausalueella olevan olion osoite (vaikka olio ei ole const-tyyppi) on vakiolauseke (kun taas paikallisen olion osoite ei ole). Nimiavaruuden viittausalueella olevan olion osoitetta voidaan siten käyttää argumenttina tyypittömälle malliparametrille. Samalla tavalla sizeof-lausekkeen tulos on vakiolauseke, jota voidaan käyttää argumenttina tyypittömälle malliparametrille:

```
template <int size> Buf{ ... };
template <int *ptr> class BufPtr { ... };

int size_val = 1024;
const int c_size_val = 1024;

Buf< 1024 > buf0; // ok
Buf< c_size_val > buf1; // ok
Buf< sizeof(size_val) > buf2; // ok: sizeof(int)
BufPtr< &size_val > bp0; // ok

// virhe: ei voi ratkaista käännöksen aikana
Buf< size_val > buf3;
```

Tässä on toinen esimerkki, jossa kuvataan, kuinka tyypitöntä malliparametria voidaan käyttää vakioarvona luokkamallin määrittelyssä ja kuinka malliargumenttia käytetään tämän malliparametrin arvon määrittämiseen:

```
template <class Type, int size>
class Fixed_Array {
public:
    Fixed_Array( Type *ar) : count( size )
    {
        for ( int ix = 0; ix < size; ++ix )
            array[ ix ] = ar[ ix ];
    }
private:
    Type array[ size ];
    int count;
};

int ia[4] = { 0, 1, 2, 3 };
Fixed_Array< int, sizeof( ia ) / sizeof( int ) > iA( ia );
```

Lausekkeita, joiden arvoista tulee yhtäsuuret, pidetään samanarvoisina malliargumentteina tyypittömälle malliparametrille. Esimerkiksi seuraavat kaikki kolme Screen-ilmentymää viittaavat samaan Screen<24,80>-malli-instantiointiin:

```
const int width = 24;
const int height = 80;
// kaikki: tyyppiä Screen< 24, 80 >
Screen< 2*12, 40*2 > scr0;
Screen< 6+6+6+6, 20*2+40 > scr1;
Screen< width, height > scr2;
```

Joitakin konversioita on sallittu malliargumentin tyypin ja tyypittömän malliparametrin välille. Sallittujen konversioiden joukko on funktion argumenteille sallittujen konversioiden alijoukko:

1. Lvalue-muunnokset, mukaan lukien lvalue rvalueksi -konversio, taulukko osoittimeksi -konversio ja funktio osoittimeksi -konversio; esimerkiksi:

```
template <int *ptr> class BufPtr { ... };

int array[10];
BufPtr< array > bpObj; // taulukko osoittimeksi -konversio
```

2. Määrekonversiot; esimerkiksi:

```
template <const int *ptr> class Ptr { ... };

int iObj;
Ptr< &iObj > pObj; // konversio int* -> const int*
```

3. Ylennykset; esimerkiksi:

```
template <int hi, int wid> class Screen { ... };

const short shi = 40;
const short swi = 132;
Screen< shi, swi > bpObj2; // ylennys short -> int
```

4. Kokonaiskonversiot; esimerkiksi:

```
template <unsigned int size> Buf{ ... };

Buf< 1024 > bObj; // konversio int -> unsigned int
```

(Näitä konversioita on käsitelty lisää kohdassa 9.3.)

Mietitäänpä esimerkiksi seuraavia esittelyitä:

```
extern void foo( char * );
extern void bar( void * );
typedef void (*PFV)( void * );

const unsigned int x = 1024;

template <class Type,
        unsigned int size,
        PFV handler> class Array { ... };

Array<int, 1024U, bar> a0; // ok: konversiota ei tarvita
Array<int, 1024U, foo> a1; // virhe: foo != PFV

Array<int, 1024, bar> a2; // ok: 1024 konvertoidaan unsigned int -tyypiksi
Array<int, 1024, foo> a3; // virhe: foo != PFV

Array<int, x, bar> a4; // ok: konversiota ei tarvita
Array<int, x, foo> a5; // virhe: foo != PFV
```

Array-luokkaoliot a0 ja a4 ovat oikein määriteltyjä, koska malliargumentit vastaavat niiden vastaavia malliparametreja täsmällisesti. Array-luokkaolio a2 on oikein määritelty, koska int-tyyppinen malliargumentti 1024 konvertoidaan tyypittömän malliparametrin size-tyyppiseksi (tyypillä unsigned int) käyttämällä kokonaiskonversiota. Array-luokkaolioiden a1, a3 ja a5 esittelyt ovat virheellisiä, koska konversiota ei ole olemassa millekään kahden funktiotyypin välille.

Seuraava konversio, joka konvertoi kokonaislukutyyppisen 0-arvon osoitintyyppiseksi arvoksi, ei ole sallittu:

```
template <int *ptr>
class BufPtr { ... };

// virhe: 0 on int-tyyppinen
// implisiittistä konversiota null-osoitinarvoksi käyttämällä
// implisiittistä osoitinkonversiota ei tehdä
BufPtr< 0 > nil;
```

Harjoitus 16.3

Yksilöi, mitkä seuraavista malli-instantioinnin käyttötilanteista saavat aikaan mallin instantioinnin, jos mitkään.

```
template < class Type >
    class Stack { };

void f1( Stack< char > ); // (a)

class Exercise {
    // ...
    Stack< double > &rsd; // (b)
```

```
    Stack< int >    si;   // (c)
};

int main() {
    Stack< char > *sc;    // (d)
    f1( *sc );            // (e)

    int iObj = sizeof( Stack< string > ); // (f)
}
```

Harjoitus 16.4

Yksilöi mitkä, jos yksikään, seuraavista malli-instantioinneista ovat kelvollisia. Perustele, miksi.

```
template < int *ptr > class Ptr { ... };
template < class Type, int size > class Fixed_Array { ... };
template < int hi, int wid > class Screen { ... };

(a) const int size = 1024;
    Ptr< &size > bp1;

(b) int arr[10];
    Ptr< arr > bp2;

(c) Ptr < 0 > bp3;

(d) const int hi = 40;
    const int wi = 80;
    Screen< hi, wi+32 > sObj;

(e) const int size_val = 1024;
    Fixed_Array< string, size_val > fa1;

(f) unsigned int fasize = 255;
    Fixed_Array< int, fasize > fa2;

(g) const double db = 3.1415;
    Fixed_Array< double, db > fa3;
```

16.3 Luokkamallien jäsenfunktiot

Aivan kuten mallittomien luokkien yhteydessä, myös luokkamallin jäsenfunktio voidaan määritellä joko luokkamallin määrittelyssä, jolloin jäsenfunktio on välitön (*inline*) tai jäsenfunktio voidaan määritellä luokkamallin määrittelyn ulkopuolelle. Olemme jo nähneet esimerkkejä välittömistä funktioista, kun esittelimme Queue-luokkamallin. Esimerkiksi Queue-muodostaja on määritelty välittömäksi luokkamallin määrittelyssä.

```
template <class Type>
class Queue {
    // ...
public:
    // välitön muodostaja-jäsenfunktio
    Queue() : front( 0 ), back ( 0 ) { }
    // ...
};
```

Luokkamallin jäsenfunktion, joka on määritelty luokkamallinsa ulkopuolelle, on käytettävä erikoissyntaksia ilmaistakseen, että se on luokkamallin jäsen. Jäsenfunktion määrittelyn alussa pitää olla avainsana `template`, jonka jälkeen tulevat malliparametrit. Esimerkiksi Queue-muodostaja olisi voitu määritellä luokkamallin määrittelyn ulkopuolelle seuraavasti:

```
template <class Type>
class Queue {
public:
    Queue( );
private:
    // ...
};
template <class Type>
inline Queue<Type>::
    Queue( ) { front = back = 0; }
```

Ensimmäisen Queue-esiintymän (sen, jonka edessä on viittausalueoperaattori `::`) jälkeen tulee malliparametriluettelo. Tämä edustaa luokkamallia, johon jäsenfunktio kuuluu. Toinen Queue-esiintymä muodostajan määrittelyssä (se, joka tulee viittausalueoperaattorin jälkeen) edustaa muodostaja-jäsenfunktion nimeä. Sen nimi voidaan mainita tai jättää mainitsematta malliparametriluettelon jälkeen. Seuraava jäsenfunktion nimi on funktiomäärittely, joka näyttää paljon mallittoman funktion määrittelyltä. Kuitenkin luokkamallin jäsenfunktion määrittely voi viitata `Type`-malliparametriin aina, kun tyyppinimeä voidaan käyttää tavallisissa funktiomäärittelyissä.

Luokkamallin jäsenfunktio on itsekin malli. C++-standardi vaatii, että sellaiset jäsenfunktiot tulisi instantioida vain, kun niitä kutsutaan tai niiden osoite otetaan. (Jotkut esistandardin toteutukset instantioivat luokkamallin jäsenfunktiot, kun itse luokkamalli instantioidaan.) Tyyppi, jota käytetään jäsenfunktion instantiointiin, on sen luokkaolion tyyppi, jolle jäsenfunktiota kutsutaan. Esimerkiksi:

```
Queue<string> qs;
```

Olio `qs` on `Queue<string>`-tyyppinen. Kun tämä luokkaolio instantioidaan, kutsutaan `Queue<string>`-luokan muodostajaa. Malliargumentti, jolla muodostaja-jäsenfunktio instantioidaan tässä tapauksessa, on string-tyyppinen.

Luokkamallin jäsenfunktiota ei instantioida automaattisesti, kun itse luokkamalli instantioidaan. Jäsenfunktio instantioidaan vain, jos ohjelma käyttää sitä (muista, että funktion kutsu tai sen osoitteen ottaminen käynnistää sen). Oikeastaan se, milloin luokkamallin jäsenfunktio instantioidaan, vaikuttaa siihen, kuinka luokkamallin jäsenfunktion määrittelyssä nimet ratkaistaan (käsitellään lisää kohdassa 16.11) ja milloin jäsenfunktion erikoistaminen voidaan esitellä (käsitellään lisää kohdassa 16.9).

16.3.1 Queue- ja QueueItem-luokkamallien jäsenfunktiomallit

Perehtyäksemme hieman enemmän siihen, kuinka luokkamallin jäsenfunktioita määritellään ja käytetään, tutkikaamme edelleen Queue-luokkamallia ja sen jäsenfunktioita.

```
template <class Type>
class Queue {
public:
    Queue() : front( 0 ), back ( 0 ) { }
    ~Queue();

    Type& remove();
    void add( const Type & );
    bool is_empty() const {
        return front == 0 ;
    }
private:
    QueueItem<Type> *front;
    QueueItem<Type> *back;
};
```

Luokkamallin määrittelyyn ei ole määritelty tuhoajaa eikä jäsenfunktioita remove() ja add(). Nämä jäsenfunktiot on määritelty luokkamallin ulkopuolelle kuten seuraavista esimerkeistä nähdään. Queue-tuhoaja tyhjentää jonon:

```
template <class Type>
Queue<Type>::~Queue()
{
    while ( ! is_empty() )
        remove();
}
```

Jäsenfunktio Queue<Type>::add() sijoittaa uuden jäsenen jonon taakse. Tässä on sen toteutus:

```
template <class Type>
void Queue<Type>::add( const Type &val )
{
    // varaa muistia uudelle QueueItem-oliolle
    QueueItem<Type> *pt =
        new QueueItem<Type>( val );

    if ( is_empty() )
        front = back = pt;
    else
    {
        back->next = pt;
        back = pt;
    }
}
```

Jäsenfunktio Queue<Type>::remove() palauttaa jäsenen arvon jonon edestä. Vastaava QueueItem-olio tuhotaan.

```
#include <iostream>
#include <cstdlib>

template <class Type>
Type Queue<Type>::remove()
{
    if ( is_empty() )
    {
        cerr << "remove() on empty queue\n";
        exit( -1 );
    }

    QueueItem<Type> *pt = front;
    front = front->next;

    Type retval = pt->item;
    delete pt;
    return retval;
}
```

Päätimme lisätä jäsenfunktioiden määrittelyt otsikkotiedostoon Queue.h ja ottaa nämä määrittelyt mukaan jokaiseen tiedostoon, jossa jäsenfunktioiden instantiointeja käytetään. (Katsomme kohdassa 16.8 sitä, miksi päätimme tehdä näin, ja vielä yleisempää kysymystä mallin käännösmallista.)

Seuraavassa ohjelmassa kuvataan, kuinka Queue-luokkamallin jäsenfunktioita voidaan käyttää ja instantioida:

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

int main()
{
    // Luokka Queue<int> instantioidaan
    // new-lauske vaatii, että Queue<int> määritellään
    Queue<int> *p_qi = new Queue<int>;

    int ival;
```

```
    for ( ival = 0; ival < 10; ++ival )
        // jäsenfunktio add() instantioidaan
        p_qi->add( ival );

    int err_cnt = 0;
    for ( ival = 0; ival < 10; ++ival ) {
        // jäsenfunktio remove() instantioidaan
        int qval = p_qi->remove();

        if ( ival != qval ) err_cnt++;
    }

    if ( !err_cnt )
        cout << "!! queue executed ok\n";
    else cerr << "?? queue errors: " << err_cnt << endl;
    return 0;
}
```

Kun ohjelma käännetään ja suoritetaan, se generoi seuraavan tulostuksen:

```
!! queue executed ok
```

Harjoitus 16.5

Käytä kohdassa 16.2 määriteltyä Screen-luokkamallia ja toteuta Screen-luokan jäsenfunktiot uudelleen jäsenfunktiomalleina, jotka toteutettiin luvun 13 kohdissa 13.3, 13.4 ja 13.6.

16.4 Ystäväesittelyt luokkamalleissa

Luokkamallissa voi esiintyä kolmenlaisia ystäväesittelyitä:

1. Malliton ystäväluokka tai -funktio. Seuraavassa esimerkissä foo()-funktio, bar()-jäsenfunktio ja foobar-luokka ovat ystäviä kaikille QueueItem-luokkamallin instantioinneille.

```
class Foo {
    void bar();
};

template <class T>
class QueueItem {
    friend class foobar;
    friend void foo();
    friend void Foo::bar();
    // ...
};
```

   Luokkaa foobar ja foo()-funktiota ei tarvitse esitellä tai määritellä globaalilla viittausalueella ennen kuin QueueItem-luokkamalli esittelee ne ystäviksi. Kuitenkin Foo-luokka pitää olla määriteltynä ennen kuin QueueItem-luokka voi esitellä sen yhden

jäsenen ystävänä. Muista, että luokan jäsenen voi esitellä vain luokkansa määrittely. QueueItem ei voi viitata Foo::bar()-jäseneen ennen kuin Foo:n luokkamäärittely on nähty.

2. *Sidottu* ystäväluokkamalli tai -funktiomalli. Seuraavassa esimerkissä on määritelty yksi yhteen -yhteys QueueItem-luokkamallin instantioinnin ja sen ystävien välille, myös malli-instantiointeihin. Jokaiselle QueueItem-luokkamallin instantioinnille ovat vastaavat yksittäiset foobar-, foo()- ja Queue::bar()-instantioinnit ystäviä.

```
template <class Type>
    class foobar{ ... };

template <class Type>
    void foo( QueueItem<Type> );

template <class Type>
class Queue {
        void bar();
        // ...
};

template <class Type>
class QueueItem {
    friend class foobar<Type>;
    friend void foo<Type>( QueueItem<Type> );
    friend void Queue<Type>::bar();

    // ...
};
```

Mallille pitää olla esittely tai määrittely ennen kuin sitä voidaan käyttää luokkamallin ystäväesittelyssä. Esimerkissämme foobar-, Queue- ja foo()-luokkamallien pitää olla esiteltyinä ennen ystäväesittelyitä QueueItem-luokassa.

Syntaksi, jota käytetään ystäväesittelyssä foo(), voi näyttää yllättävältä:

```
friend void foo<Type>( QueueItem<Type> );
```

Funktion nimen jälkeen tulee luettelo eksplisiittisiä malliargumentteja: foo<Type>. Tätä syntaksia käytetään, kun halutaan määrittää, että ystäväesittely viittaa foo()-funktiomallin instantiointiin. Jos eksplisiittiset malliargumentit olisi jätetty pois kuten seuraavassa

```
friend void foo( QueueItem<Type> );
```

silloin ystäväesittelyn olisi tulkittu viittaavan mallittomaan funktioon parametrityypillä, joka on QueueItem-luokkamallin ilmentymä. Kuten mainittiin kohdassa 10.6, samanniminen funktiomalli ja malliton funktio voivat esiintyä yhtä aikaa eikä funktiomallin esittelyn läsnäolo ennen QueueItem-luokan määrittelyä pakota

ystäväesittelyä viittaamaan tähän malliin. Jotta ystäväesittely saataisiin viittaamaan funktiomallin instantiointiin, pitää sille määrittää eksplisiittinen malliargumenttiluettelo.

3. *Sitomaton* ystäväluokkamalli tai -funktiomalli. Seuraavassa esimerkissä on määritelty yksi moneen -yhteys QueueItem-luokkamallin instantioinnin ja ystävän välille. Jokaiselle QueueItem-luokkamallin tyyppi-instantioinnille ovat kaikki foobar-, foo()- ja Queue<T>::bar()-instantioinnit ystäviä.

```
template <class Type>
class QueueItem {
    template <class T>
        friend class foobar;

    template <class T>
        friend void foo( QueueItem<T> );

    template <class T>
        friend void Queue<T>::bar();

    // ...
};
```

Tulisi panna merkille, että luokkamallien tämänkaltaisia ystäväesittelyitä ei tueta toteutuksissa, jotka tukevat C++-esistandardia.

16.4.1 Queue- ja QueueItem-luokkamallien ystäväesittelyt

Koska QueueItem-luokkamallia ei aiota käyttää pääohjelmasta, on QueueItem-muodostaja siirretty QueueItem-luokkamallin yksityiseen osaan. Queue pitää nyt esitellä QueueItem:in ystävänä, jotta QueueItem-luokkaolioita voitaisiin luoda ja käsitellä.

On olemassa kaksi menetelmää, joilla luokkamalli voidaan esitellä ystäväksi. Ensimmäinen menetelmä on esitellä kaikki mahdolliset Queue-ilmentymät jokaisen QueueItem-instantioinnin ystäviksi:

```
template <class Type>
class QueueItem {
    // kaikki Queue-instantioinnit ovat ystäviä
    // jokaiselle QueueItem-instantioinnille
    template <class T> friend class Queue;
};
```

Tämä ei kuitenkaan ole aivan suunnittelun tarkoitus. Ei ole järkeä esimerkiksi, että string-tyypillä instantioidusta Queue:sta tehdään complex<double>-tyypillä instantioidun Queue:n ystävä. Queue<string>:in tulisi olla ystävä vain merkkijonoilla instantioiduille QueueItem-ilmentymille. Haluamme siis yksi yhteen -yhteyden Queue- ja QueueItem-ilmentymien välille jokaiselle instantioidulle tyypille. Tämä saavutetaan käyttämällä toisenlaista ystäväesittelyä:

```
template <class Type>
class QueueItem {
    // jokaiseen QueueItem-instantiointiin liittyy
    // Queue-instantiointi sen ystävänä
    friend class Queue<Type>;

    // ...
};
```

Tämä esittely ilmaisee, että jokaiselle tietyntyyppiselle QueueItem-instantioinnille on vastaava Queue-instantiointi sen ystävänä. Tarkoittaa, että int-tyyppinen Queue-instantiointi on int-tyyppisen QueueItem-instantioinnin ystävä. Se ei ole sellaisten QueueItem-instantiointien ystävä, joiden tyyppi on complex<double> tai string.

Missä tahansa kohdassa voi käyttäjä haluta näyttää Queue-olion sisällön. Eräs menetelmä tämän mahdollistamiseksi on tehdä ylikuormitettu ilmentymä tulostusoperaattorista. Tämä operaattori pitää esitellä Queue-luokkamallin ystäväfunktiona, koska sen täytyy käsitellä luokan yksityisiä jäseniä. Miltä operaattorin tunnisteen tulisi näyttää?

```
// minkä muotoinen Queue-argumentti?
ostream& operator<<( ostream &, ??? );
```

Koska Queue on luokkamalli, pitää mallin instantioinnin nimen määrittää koko argumenttiluettelo. Esimerkiksi:

```
ostream& operator<<( ostream &, const Queue<int> & );
```

Tämä määrittelee tulostusoperaattorin Queue-luokkamallin instantioinnille, jonka jäsenten tyyppi on int. Mitä sitten, jos Queue muodostuu string-tyyppisistä jäsenistä?

```
ostream& operator<<( ostream &, const Queue<string> & );
```

Sen sijaan, että määrittelisimme jokaisen tietyn tulostusoperaattorin eksplisiittisesti tarpeen mukaan, määrittelemme yleisen tulostusoperaattorin, jolla voidaan käsitellä kaikki Queue-instantioinnit. Esimerkiksi:

```
ostream& operator<<( ostream &, const Queue<Type> & );
```

Jotta tämä kuitenkin toimisi, pitää tästä ylikuormitetusta tulostusoperaattorista tehdä vuorostaan funktiomalli:

```
template <class Type> ostream&
    operator<<( ostream &, const Queue<Type> & );
```

Kun se on tehty, joka kerta, kun Queue-instantiointi on välitetty ostream-virtaan, funktiomalli instantioidaan ja käynnistetään. Tässä on eräs mahdollinen toteutus tulostusoperaattorista funktiomallina:

```
template <class Type>
ostream& operator<<( ostream &os, const Queue<Type> &q )
{
    os << "< ";
    QueueItem<Type> *p;
```

```
    for ( p = q.front; p; p = p->next )
            os << *p << " ";
    os << " >";
    return os;
}
```

Jos int-tyyppiset Queue-oliot sisältävät arvot 3, 5, 8 ja 13, tämän Queue:n tulostus näyttää seuraavalta:

```
< 3 5 8 13 >
```

Huomaa, että tulostusoperaattori viittaa Queue:n yksityiseen front-jäseneen. Seuraava asia, joka meidän pitää tehdä, on esitellä operaattori Queue:n ystävänä, kuten seuraavassa:

```
template <class Type>
class Queue {
    friend ostream&
        operator<<( ostream &, const Queue<Type> & );
    // ...
};
```

Huomaa, että aivan kuten aikaisemmin Queue-luokkamallin ystäväesittelyssä, tämä esittely luo yksi yhteen -yhteyden Queue-instantioinnin ja sitä vastaavan operator<<()-ilmentymän kanssa.

Queue-elementtien varsinainen tulostus luottaa QueueItem:in tulostusoperaattoriin operator<<():

```
os << *p;
```

QueueItem:in tulostusoperaattori pitää myös toteuttaa funktiomallina. Tämä varmistaa, että sopiva operator<<()-operaattori instantioidaan automaattisesti tarvittaessa:

```
template <class Type>
ostream& operator<<( ostream &os, const QueueItem<Type> &qi )
{
    os << qi.item;
    return os;
}
```

Koska tämä operaattori käsittelee QueueItem:in yksityistä item-jäsentä, pitää tämä operaattori esitellä QueueItem-luokkamallin ystävänä. Se voidaan tehdä seuraavasti:

```
template <class Type>
class QueueItem {
    friend class Queue<Type>;
    friend ostream&
        operator<<( ostream &, const QueueItem<Type> & );
    // ...
};
```

QueueItem:in tulostusoperaattori operator<<() luottaa, että item itse hoitaa varsinaisen tulostuksen:

```
os << qi.item;
```

Tämä tuo esille hienoisen tyyppiriippuvuuden Queue-instantioinnista. Itse asiassa jokaisella Queue:hun sidotulla käyttäjän määrittelemällä luokkatyypillä, joka aikoo tulostaa itsensä, pitää olla tulostusoperaattori. Kielessä ei ole mekanismia, jolla joko määrittäisi tai pakottaisi tuon riippuvuuden itse Queue-luokkamallin määrittelyyn. Sen sijaan, jos tulostusoperaattoria ei ole määritelty tyypille, jota käytetään Queue-mallin instantiointiin, ja yritetään tulostaa instantioinnin sisältö, saadaan aikaan käännösvirhe siinä kohdassa, jossa kelpaamatonta tulostusoperaattoria on käytetty. Queue voidaan instantioida tyypillä, jossa ei ole määritelty tulostusoperaattoria — edellyttäen, että ei yritetä tulostaa Queue:n sisältöä.

Seuraavassa ohjelmassa kuvataan, kuinka Queue- ja QueueItem-luokkamallien ystäväfunktioita voitaisiin instantioida ja käyttää:

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

int main() {
    Queue<int> qi;
    // instantioi molemmat ilmentymät:
    //   ostream& operator<<(ostream &os, const Queue<int> &)
    //   ostream& operator<<(ostream &os, const QueueItem<int> &)
    cout << qi << endl;

    int ival;
    for ( ival = 0; ival < 10; ++ival )
        qi.add( ival );
    cout << qi << endl;

    int err_cnt = 0;
    for ( ival = 0; ival < 10; ++ival ) {
        int qval = qi.remove();
        if ( ival != qval ) err_cnt++;
    }

    cout << qi << endl;
    if ( !err_cnt )
        cout << "!! queue executed ok\n";
    else cout << "?? queue errors: " << err_cnt << endl;
    return 0;
}
```

Kun ohjelma käännetään ja suoritetaan, se generoi seuraavan tulostuksen:

```
< >
< 0 1 2 3 4 5 6 7 8 9 >
< >
!! queue executed ok
```

Harjoitus 16.6

Käytä Screen-luokkamallia, joka määriteltiin harjoituksessa 16.5, ja toteuta malleina uudelleen syöttö- ja tulostusoperaattorit, jotka määriteltiin Screen-luokalle harjoituksessa 15.6 kohdassa 15.2. Perustele syyt valitsemiisi ystäväesittelyihin, jotka lisäät Screen-luokkamalliin.

16.5 Luokkamallien staattiset tietojäsenet

Luokkamallissa voidaan esitellä staattisia tietojäseniä. Jokaisella luokkamallin instantioinnilla on omat staattiset tietojäsenensä. Jotta voisimme kuvata tätä, esitelkäämme new()- ja delete()-operaattorit QueueItem-luokkamalliin. Jotta tämä voitaisiin tehdä, pitää lisätä kaksi staattista tietojäsentä QueueItem-luokkamalliin:

```
static QueueItem<Type> *free_list;
static const unsigned QueueItem_chunk;
```

QueueItem-luokkamallin muokkaukset näyttävät tältä:

```
#include <cstddef>

template <class Type>
class QueueItem {
    // ...
private:
    void *operator new( size_t );
    void operator delete( void *, size_t );
    // ...
    static QueueItem *free_list;
    static const unsigned QueueItem_chunk;
    // ...
};
```

new()- ja delete()-operaattorit on esitelty yksityisinä, jotta estettäisiin pääohjelmaa luomasta QueueItem-tyyppisiä olioita vapaavarastoon. Vain QueueItem:in jäsenet ja ystävät (kuten Queue-malli) voivat luoda (ja poistaa) QueueItem-tyyppisiä olioita vapaavarastoon.

new()-operaattorin määrittelyn toteutus voisi näyttää tältä:

```
template <class Type> void*
    QueueItem<Type>::operator new( size_t size )
{
    QueueItem<Type> *p;
    if ( ! free_list )
    {
        size_t chunk = QueueItem_chunk * size;
        free_list = p =
            reinterpret_cast< QueueItem<Type>* >
                    ( new char[chunk] );

        for ( ; p != &free_list[ QueueItem_chunk - 1 ]; ++p )
            p->next = p + 1;
        p->next = 0;
    }

    p = free_list;
    free_list = free_list->next;
    return p;
}
```

Tässä on delete()-operaattorimallin toteutus:

```
template <class Type>
void QueueItem<Type>::
    operator delete( void *p, size_t )
{
    static_cast< QueueItem<Type>* >( p )->next = free_list;
    free_list = static_cast< QueueItem<Type>* > ( p );
}
```

Kaikki, mitä jää jäljelle tehtäväksi, on staattisten jäsenten, free_list ja QueueItem_chunk, alustus. Staattisten tietojäsenten määrittelyn mallimuoto on seuraava:

```
/* generoi jokaiselle QueueItem-instantioinnille
 * vastaava free_list, alusta se arvolla 0
 */
template <class T>
    QueueItem<T> *QueueItem<T>::free_list = 0;

/* generoi jokaiselle QueueItem-instantioinnille
 * vastaava QueueItem_chunk, alusta se arvolla 24
 */
template <class T>
    const unsigned int
    QueueItem<T>::QueueItem_chunk = 24;
```

Staattisen tietojäsenen mallimäärittelyn pitää esiintyä luokkamallimäärittelyn ulkopuolella. Tästä syystä mallimäärittely alkaa avainsanalla template, jonka jälkeen tulee luokkamallin parametriluettelo <class T>. Staattisen tietojäsen nimi varustetaan QueueItem<T>::-etuliitteellä, joka ilmaisee, että jäsen kuuluu QueueItem-luokkamalliin. Staattisten tietojäsenten määrittelyt lisätään otsikkotiedostoon Queue.h. Nämä määrittelyt pitää ottaa mukaan tiedostoihin, joissa staattisten tietojäsenten instantiointeja käytetään. (Katsomme sitä, miksi päätimme tehdä näin, ja tähän liittyvää mallin käännösmallia kohdassa 16.8.)

Staattinen tietojäsen instantioidaan mallimäärittelystä vain, jos sitä käytetään ohjelmassa. Luokkamallin staattinen tietojäsen on itsekin malli. Staattisen tietojäsenen mallimäärittely ei saa aikaan muistin varaamista. Muistia varataan vain staattisen tietojäsenen tietyille instantioinneille. Jokaisen staattisen tietojäsenen instantiointi vastaa luokkamallin instantiointia. Staattisen tietojäsenen instantiointiin viitataan siten tietyn luokkamallin instantioinnin kautta. Esimerkiksi:

```
// virhe: QueueItem ei ole varsinainen instantiointi
int ival0 = QueueItem::QueueItem_chunk;

int ival1 = QueueItem<string>::QueueItem_chunk; // ok
int ival2 = QueueItem<int>::QueueItem_chunk;    // ok
```

Harjoitus 16.7

Käytä kohdassa 15.8 määriteltyjä operaattoreita new() ja delete() sekä niihin liittyviä staattisia jäseniä screenChunk ja freeStore käyttäen ja toteuta nämä operaattorit sekä staattiset jäsenet harjoituksessa 16.6 määritellylle Screen-luokkamallille.

16.6 Luokkamallien sisäkkäiset tyypit

QueueItem-luokkamalli on suunniteltu toimimaan vain Queue:n toteutuksessa. Tämän pakottamiseksi QueueItem-luokkamallilla on yksityinen muodostaja, joka mahdollistaa sen Queue-ystäväluokkamallin jäsenfunktioiden, mutta ei muiden luokkien tai funktioiden (paitsi sen omien jäsenfunktioiden) luoda QueueItem-tyyppisiä olioita. Vaikka QueueItem on luokkamalli, joka näkyy koko ohjelmaan, ei ohjelma pysty joko luomaan QueueItem-olioita tai viittaamaan yhteenkään QueueItem:in jäseneen kutsumatta Queue:n jäsenfunktioita.

Vaihtoehtoinen toteutusstrategia on laittaa QueueItem-luokkamallin määrittelyt sisäkkäin Queue-luokkamallin yksityiseen osaan. Kun QueueItem on sisäkkäinen yksityinen tyyppi, ei siihen pääse käsiksi pääohjelmasta. Ja koska se on yksityinen sisäkkäinen tyyppi, vain Queue-luokkatyyppi ja sen ystävä — tulostusoperaattori — voivat käsitellä sitä. Jos teemme QueueItem:in jäsenistä julkisia, ei ole enää välttämätöntä esitellä Queue:ta QueueItem:in ystävänä.

Tämä toteutus säilyttää alkuperäisen toteutuksemme ajatuksen ja mallintaa elegantimmin QueueItem- ja Queue-luokkamallien suhdetta toisiinsa.

Koska Queue vaatii vastaavan QueueItem-luokan jokaiselle tyypille, josta se on instantioitu, sisäkkäinen luokka on myös luokkamalli. Luokkamallien sisäkkäiset luokat ovat au-

tomaattisesti luokkamalleja ja ympäröivän luokkamallin malliparametria voidaan käyttää sisäkkäiselle luokkamallille. Esimerkiksi:

```
template <class Type>
class Queue {
    // ...
private:
    class QueueItem {
    public:
        QueueItem( Type val )
            : item( val ), next( 0 ) { ... }

        Type item;
        QueueItem *next;
    };
    // koska QueueItem on sisäkkäinen tyyppi
    // eikä malli, joka on määritelty Queue:n ulkopuolelle,
    // voidaan malliargumentti <Type> jättää pois
    // QueueItem:in jälkeen
    QueueItem *front, *back;
    // ...
};
```

Jokainen Queue-instantiointi generoi oman QueueItem-luokan Type:lle sopivin malliargumentein. QueueItem-luokkamallin instantioinnin ja ympäröivän Queue-luokkamallin instantioinnin välinen yhteys on yksi yhteen.

Luokkamallin sisäkkäistä luokkaa ei instantioida automaattisesti, kun ympäröivä luokkamalli instantioidaan. Sisäkkäinen luokka instantioidaan vain, jos sitä itseään käytetään yhteydessä, joka vaatii täydellisen luokkatyypin. Mainitsimme esimerkiksi kohdassa 16.2, että jos Queue-luokkamalli instantioidaan int-tyypillä, ei QueueItem<int>-tyyppiä instantioida automaattisesti. Jäsenet front ja back ovat osoittimia QueueItem<int>-tyyppi, eikä ole välttämätöntä instantioida QueueItem<int>-tyyppiä, jos on esitelty vain osoittimia tähän luokkatyyppiin. Se, että tehdään QueueItem:istä Queue-luokkamallin sisäkkäinen luokka, ei muuta tätä. QueueItem<int> instantioidaan silti yhä vain, kun front- ja back-jäseniin viitataan käänteisesti Queue<int>-luokan jäsenfunktioissa.

Luokkamalliin voidaan esitellä myös lueteltuja joukkoja ja typedef-nimiä. Esimerkiksi:

```
template <class Type, int size>
class Buffer {
public:
    enum Buf_vals { last = size-1, Buf_size };
    typedef Type BufType;
    BufType array[ size ];
    // ...
};
```

Sen sijaan, että Buffer-luokkamallissa tehtäisiin eksplisiittinen Buf_size-tietojäsen, esitellään siinä lueteltu tyyppi pareine jäsenineen malliparametrilla alustettuna. Esimerkiksi esittely

```
Buffer<int, 512> small_buf;
```

asettaa sen Buf_size:n arvoksi 512 ja last:in arvoksi 511. Samalla tavalla esittely

```
Buffer<int, 1024> medium_buf;
```

asettaa sen Buf_size:n arvoksi 1024 ja last:in arvoksi 1023.

Julkista sisäkkäistä tyyppiä voidaan käyttää luokkamäärittelynsä ulkopuolella. Kuitenkin pääohjelma voi viitata luokkamallin vain julkisen sisäkkäisen tyypin instantiointiin (tai sisäkkäisen luetellun joukon jäseneen). Tässä tapauksessa sisäkkäisen tyypin nimen eteen pitää laittaa luokkamallin instantioinnin nimi. Esimerkiksi:

```
// virhe: mikä Buffer:in instantiointi?
Buffer::Buf_vals bfv0;

Buffer<int,512>::Buf_vals bfv1; // ok
```

Tämä sääntö käy myös, jos sisäkkäinen tyyppi ei käytä jotain ympäröivän luokkamallin parametreista. Esimerkiksi:

```
template <class T> class Q {
public:
    enum QA { empty, full }; // muuttumaton
    QA status;
    // ...
};

#include <iostream>

int main() {
    Q<double> qd;
    Q<int> qi;

    qd.status = Q::empty; // virhe: mikä Q:n instantiointi?
    qd.status = Q<double>::empty; // ok

    int val1 = Q<double>::empty;
    int val2 = Q<int>::empty;
    if ( val1 != val2 )
        cerr << "implementation error!" << endl;
    return 0;
}
```

Vaikka empty:n arvo on sama jokaisessa Q:n instantioinnissa, pitää empty:yn viittaavan koodin määrittää, mihin tiettyyn Q:n ilmentymään luetellun joukon jäsen kuuluu.

Harjoitus 16.8

Määrittele kohdassa 13.10 määritelty List-luokka ja sen sisäkkäinen ListItem-luokka luokkamalleina. Tee myös mallimäärittelyt vastaaville luokan jäsenille.

16.7 Jäsenmallit ◆

Funktio- tai luokkamalli voi olla tavallisen luokan tai luokkamallin jäsen. Jäsenmallin (*member template*) määrittely näyttää mallimäärittelyltä: jäsenmäärittelyn edessä on avainsana template, jonka jälkeen tulee malliparametriluettelo. Esimerkiksi:

```
template <class T>
class Queue {
private:
    // luokkajäsenmalli
    template <class Type>
        class CL
    {
        Type member;
        T mem;
    };
    // ...
public:
    // funktiojäsenmalli
    template <class Iter>
        void assign( Iter first, Iter last )
    {
        while ( ! is_empty() )
            remove();  // kutsuu: Queue<T>::remove()

        for ( ; first != last; ++first )
            add( *first );  // kutsuu: Queue<T>::add( const T & )
    }
};
```

(Huomaa, että esistandardin C++-toteutukset eivät tue jäsenmalleja. Tämä piirre lisättiin C++:aan tukemaan luvussa 6 esiteltyjä säiliöluokka-abstrakteja kuten kerromme seuraavissa kappaleissa.)

Jäsenmallin esittelyssä on omat malliparametrit. Esimerkiksi luokkajäsenmallilla CL on oma malliparametrinsa nimeltään Type ja funktiojäsenmallilla assign() on oma malliparametrinsa Iter. Lisäksi jäsenmallin määrittelyssä voidaan käyttää myös ympäröivän luokkamallin malliparametreja. Esimerkiksi luokkajäsenmallilla CL on T-tyyppinen tietojäsen, joka on ympäröivän Queue-luokkamallin malliparametri.

Kun Queue-luokkamalliin esitellään jäsenmalli, se tarkoittaa, että Queue:n instantiointi sisältää potentiaalisesti loputtoman määrän sisäkkäisiä CL-luokkia ja assign()-jäsenfunktioita. Esimerkiksi Queue<int>-instantiointi voi sisältää seuraavat sisäkkäiset tyypit:

```
Queue<int>::CL<char>
Queue<int>::CL<string>
```

Samalla tavalla Queue<int> voi sisältää seuraavat jäsenfunktiot:

```
void Queue<int>::assign( int *, int * )
```

```
void Queue<int>::assign( vector<int>::iterator,
                         vector<int>::iterator )
```

Jäsenmalli noudattaa samoja käsittelysääntöjä kuin muutkin luokkajäsenet. Koska luokka-jäsenmalli CL on Queue-luokkamallin yksityinen jäsen, vain Queue:n jäsenfunktiot ja ystävät voivat viitata luokkajäsenmallin instantiointeihin. Koska funktiojäsenmalli assign() on julkinen jäsen, voidaan sitä käyttää koko ohjelman alueella.

Jäsenmalli instantioidaan vain, kun sitä itseään käytetään ohjelmassa. Esimerkiksi assign() instantioidaan, kun sitä käytetään main()-funktiossa, seuraavasti:

```
int main()
{
    // instantioidaan Queue<int>
    Queue<int> qi;

    // instantioidaan Queue<int>::assign( int *, int * )
    int ai[4] = { 0, 3, 6, 9 };
    qi.assign( ai, ai + 4 );

    // instantioidaan Queue<int>::assign( vector<int>::iterator,
    //                                    vector<int>::iterator)
    vector<int> vi( ai, ai + 4 );
    qi.assign( vi.begin(), vi.end() );

}
```

Queue-luokkamallin funktiojäsenmalli assign() on hyvä esimerkki siitä, miksi jäsenmalleja tarvitaan säiliötyyppien tukemiseen. Voisimme esimerkiksi lisätä annettuun jonotyyppiin Queue<int> minkä tahansa muun säiliön sisällön (listan, vektorin tai yksinkertaisen taulukon sisällön) elementeillä, jotka ovat joko int-tyyppiä (elementtien tyyppi jonossa) tai tyyppiä, joka voidaan konvertoida int-tyypiksi. Jäsenmalli assign() mahdollistaa, että voimme tehdä juuri sen. Koska voidaan käyttää mitä tahansa säiliötyyppiä, ohjelmoimme assign()-jäsenmallin rajapinnan käyttämään iteraattoreita ja siten eristämme sen toteutuksen todellisesta säiliötyypistä, johon iteraattorit viittaavat.

main()-funktiossa instantioidaan ensiksi assign()-jäsenmalli tyypillä int*, joka mahdollistaa int-taulukon sisällön sijoituksen qi:hin. Sitten jäsenmalli instantioidaan tyypillä vector<int>::iterator, joka mahdollistaa int-vektorin sisällön sijoituksen qi:hin. Jonoon sijoitettavan säiliön ei tarvitse sisältää int-tyyppisiä elementtejä. Kaikki tyypit, jotka voidaan konvertoida int-tyypeiksi, ovat myös kelvollisia. Jotta voisimme perustella miksi, katsokaamme assign()-määrittelyä:

```
template <class Iter>
    void assign( Iter first, Iter last )
{
    // poista jäsenet Queue:ta varten

    for ( ; first != last; ++first )
        add( *first );
```

```
}
```

Funktio add(), jota assign() kutsuu, on jäsenfunktio Queue<Type>::add(). Kun Queue instantioidaan int-tyypillä, tällä jäsenfunktiolla on seuraava prototyyppi:

```
void Queue<int>::add( const int &val )
```

Argumentin *first pitää olla joko int-tyyppinen tai tyyppi, joka voi alustaa viittaustyyppisen parametrin const int -tyyppiin. Tyyppikonversiot ovat sallittuja. Jos käytämme uudelleen SmallInt-luokkaa, joka määriteltiin kohdassa 15.9, voidaan SmallInt-tyyppisten elementtien säiliö sijoittaa Queue<int>-tyyppiseen jonoon käyttämällä funktiojäsenmallia assign(). Tämä on mahdollista, koska SmallInt-luokka sisältää konversiofunktion, jolla voidaan konvertoida SmallInt-tyyppinen arvo int-tyyppiseksi arvoksi:

```
class SmallInt {
public:
    SmallInt( int ival = 0 ) : value( ival ) { }

    // konversiofunktio: SmallInt ==> int
    operator int() { return value; }

    // ...
private:
    int value;
};

int main()
{
    // instantioidaan Queue<int>
    Queue<int> qi;

    vector<SmallInt> vsi;
    // aseta vektorin sisältö

    // instantioidaan:
    // Queue<int>::assign( vector<SmallInt>::iterator,
    //                     vector<SmallInt>::iterator )
    qi.assign( vsi.begin(), vsi.end() );

    list<int*> lpi;
    // aseta listan sisältö

    // virhe, kun jäsenmalli assign() instantioidaan
    // ei konversiota int* ==> int
    qi.assign( lpi.begin(), lpi.end() );
}
```

Ensimmäinen assign()-instantiointi on kelvollinen, koska on olemassa implisiittinen konversio SmallInt-tyypistä int-tyypiksi, joten kutsu add()-funktioon ensimmäisessä assign()-instantioinnissa on kelvollinen. Toinen instantiointi on virheellinen, koska int*-tyyppistä oliota ei voi alustaa viittauksella const int -tyyppiin. Kutsu add()-funktioon toisessa assign()-instantioinnissa on virheellinen.

C++-vakiokirjastoon määritellyillä säiliötyypeillä on funktiojäsenmalli nimeltään assign(), joka käyttäytyy täsmälleen samalla tavalla kuin Queue-luokkamallimme assign()-jäsenmalli.

Mikä tahansa jäsenfunktio voidaan määritellä jäsenmalliksi. Esimerkiksi muodostaja voidaan määritellä jäsenmallina. Voimme määritellä sellaisen muodostajan Queue-luokkamallillemme seuraavasti:

```
template <class T>
class Queue {
    // ...
public:
    // muodostajajäsenmalli
    template <class Iter>
    Queue( Iter first, Iter last )
        : front( 0 ), back( 0 )
    {
        for ( ; first != last; ++first )
            add( *first );
    }
};
```

Tällainen muodostaja mahdollistaa jonon alustamisen toisen säiliön sisällöllä. C++-vakiokirjastoon määritellyillä säiliötyypeillä on myös muodostajajäsenmalleja, jotka mahdollistavat niiden alustamisen minkä tahansa säiliön sisällöllä. Itse asiassa tämän kohdan ensimmäisessä main()-funktion määrittelyssä käytetään vektorin muodostajajäsenmallia:

```
vector<int> vi( ai, ai + 4 );
```

Tämä määrittely instantioi vector<int>-säiliön muodostajajäsenmallin tyypillä int*, joka mahdollistaa vektorin alustamisen int-tyyppisen taulukon elementtien sisällöllä.

Kuten mallittomat jäsenet, myös jäsenmalli voidaan määritellä sitä ympäröivän luokan tai luokkamallin määrittelyn ulkopuolelle. Esimerkiksi luokkajäsenmalli CL tai jäsenfunktiomalli assign() voidaan määritellä Queue-luokkamallin ulkopuolelle kuten seuraavassa:

```
template <class T>
class Queue {
private:
    template <class Type> class CL;
    // ...
public:
    template <class Iter>
        void assign( Iter first, Iter last );
    // ...
};
```

```
template <class T> template <class Type>
    class Queue<T>::CL<Type>
{
    Type member;
    T mem;
};

template <class T> template <class Iter>
    void Queue<T>::assign( Iter first, Iter last )
{
    while ( ! is_empty() )
        remove();

    for ( ; first != last; ++first )
        add( *first );
}
```

Jäsenmallin määrittelyyn, joka on luokkamallin määrittelyn ulkopuolella, pitää liittää sen eteen luokkamallin malliparametriluettelo ja sen jälkeen jäsenen oma malliparametriluettelo. Tästä syystä assign()-jäsenfunktiomallin määrittely alkaa näin

```
template <class T> template <class Iter>
```

Ensimmäinen malliparametriluettelo, template<class T>, kuuluu Queue-luokkamallille. Toinen malliparametriluettelo, template<class Iter>, kuuluu assign()-jäsenmallille. Malliparametrien ei tarvitse olla samannimisiä kuin ne, jotka on määritetty luokkamallin määrittelyssä. Esimerkiksi seuraava silti määrittelee Queue-luokkamallin assign()-funktiojäsenmallin:

```
template <class TT> template <class IterType>
    void Queue<TT>::assign( IterType first, IterType last )
    { ... }
```

16.8 Luokkamallit ja käännösmalli ◆

Luokkamallin määrittely toimii vain ohjeena loputtomalle luokkatyyppien määrittelylle. Itsessään mallimäärittely ei määrittele luokkatyyppiä. Kun esimerkiksi kääntäjä näkee luokkamallin määrittelyn

```
template <class Type>
    class Queue { ... };
```

se tallentaa Queue:n sisäisen esitystavan. Myöhemmin, kun kääntäjä näkee todellisen instantioinnin tästä luokkamallista kuten

```
int main() {
    Queue<int> *p_qi = new Queue<int>;
}
```

se instantioi luokkatyypin Queue<int> käyttäen sisäisesti tallennettua esitystapaa Queue:n mallimäärittelystä.

Luokkamalli instantioidaan vain, jos sitä on käytetty yhteydessä, joka vaatii täydellisen luokkamäärittelyn. (Tätä käsiteltiin yksityiskohtaisemmin kohdassa 16.2.) Edellisessä esimerkissä luokkamalli Queue<int> instantioidaan, koska kääntäjän pitää tietää luokkatyypin Queue<int> koko, jotta se voi varata oikean määrän muistia oliolle, joka luodaan new-lausekkeella.

Kääntäjä voi instantioida luokkamallin vain, jos luokkamallin varsinainen määrittely eikä vain esittely on nähty. Luokkamallin määrittelyn pitää olla olemassa, kun mallia käytetään tavalla, joka vaatii sen instantioinnin:

```
// luokkamallin esittely
template <class Type>
    class Queue;

Queue<int>* global_pi = 0; // ok: luokkamäärittelyä ei tarvita

int main() {
    // virhe: instantiointi tarvitaan
    //      luokkamallimäärittelyn pitää olla näkyviss'
    Queue<int> *p_qi = new Queue<int>;
}
```

Luokkamalli voidaan instantioida samalla tyypillä useammassa kuin yhdessä tiedostossa. Kuten on asianlaita luokkatyyppien yhteydessä, jolloin luokkamäärittely pitää olla tehtynä jokaisessa tiedostossa, jossa luokkajäseniä käytetään, kääntäjä instantioi luokkamallin tietylle tyypille jokaisessa tiedostossa, jossa tätä instantiointia käytetään yhteydessä, joka vaatii täydellisen luokkamäärittelyn. Jotta voisit varmistua siitä, että luokkamallin määrittely on käytettävissä jokaisessa tiedostossa, jossa se täytyy instantioida, tulisi luokkamallin määrittelyt sijoittaa otsikkotiedostoihin.

Jäsenfunktiot, staattiset tietojäsenet ja luokkamallien sisäkkäiset tyypit käyttäytyvät melko samalla tavalla kuin itse mallit. Luokkamallin jäsenten määrittelyitä käytetään generoitaessa jäsenilmentymiä jokaiselle tietylle luokkamallin instantioinnille. Kun esimerkiksi kääntäjä näkee jäsenfunktion määrittelyn

```
template <class Type>
void Queue<Type>::add( const Type &val )
    { ... }
```

se tallentaa Queue<Type>::add():n sisäisen esitystavan. Myöhemmin, kuin kääntäjä näkee todellisen käyttötilanteen tästä jäsenfunktiosta, esimerkiksi Queue<int>-tyyppisen olion kautta, Queue<int>::add(const int &) instantioidaan tallennetun jäsenfunktion määrittelyn sisäisen esitystavan avulla:

```
#include "Queue.h"
```

```
int main() {
    // instantioidaan Queue<int>
    Queue<int> *p_qi = new Queue<int>;
    int ival;
    // ...
    // instantioidaan Queue<int>::add( const int & )
    p_qi->add( ival );
    // ...
}
```

Kun luokkamalli instantioidaan tietyllä tyypillä, se ei saa aikaan luokkamallin jäsenten määrittelyiden samanlaisten tyyppien instantiointia automaattisesti. Jäsenet instantioidaan vain, jos niitä itseään käytetään ohjelmassa tavalla, joka vaatii määrittelyn olevan tunnettu (eli jos sisäkkäistä tyyppiä käytetään tavalla, joka vaatii täydellisen luokkatyypin; jos jäsenfunktiota kutsutaan; jos jäsenen osoite otetaan tai jos staattisen tietojäsenen arvoa tutkitaan).

Jäsenfunktioiden ja luokkamallien staattisten tietojäsenten instantioinnit nostavat esille samanlaisia kysymyksiä kuin ne, joita käsittelimme funktiomallien yhteydessä kohdassa 10.5: jotta kääntäjä kykenisi instantioimaan jäsenfunktion tai luokkamallin staattisen jäsenen, pitääkö jäsenen määrittelyn olla näkyvissä, kun jotain niiden instantioinneista käytetään? Pitääkö esimerkiksi `add()`-jäsenfunktion määrittelyn esiintyä ennen kuin sen kokonaislukuilmentymää kutsutaan `main()`-funktiossa? Sijoitammeko jäsenfunktioiden määrittelyt ja luokkamallien staattiset tietojäsenet otsikkotiedostoihin (kuten teemme välittömille jäsenfunktiomäärittelyille), jotta ne voidaan ottaa mukaan kaikkialle, jossa jotain niiden instantiointeja käytetään? Vai onko se luokkamallin määrittelyn instantiointi tarpeeksi, jolla mahdollistetaan käyttäjien käyttää näitä jäseniä, kun sallimme jäsenmäärittelyiden sijoittamisen tekstitiedostoihin (jonne tavallisesti laitamme muiden kuin välittömien jäsenfunktioiden määrittelyt ja luokkatyyppiset staattiset tietojäsenet)?

Jotta voisimme vastata näihin kysymyksiin, pitää meidän palata aiheeseen C++:n *mallin käännösmalli* (*template compilation model*), joka määrittää vaatimukset siitä, malleja määrittelevät ja käyttävät ohjelmat tulee järjestää. Kohdassa 10.5 kuvatut kaksi käännösmallia — mukaan ottava ja erotteleva käännösmalli — käyvät myös jäsenfunktioiden ja luokkamallien staattisten tietojäsenten määrittelyihin. Tämän kohdan loppuun saakka kuvataan näitä käännösmalleja ja sitä, kuinka niitä käytetään näiden jäsenmäärittelyiden kanssa.

16.8.1 Mukaan ottava käännösmalli

Mukaan ottavassa käännösmuodossa pitää jäsenfunktioiden ja luokkamallin staattisten tietojäsenten määrittelyt ottaa mukaan jokaiseen tiedostoon, jossa niitä instantioidaan. Tämä tapahtuu automaattisesti välittömille jäsenfunktioille, jotka on määritelty välittömiksi luokkamallin määrittelyssä. Jos kuitenkin jäsenfunktio on määritelty luokkamallin määrittelyn ulkopuolelle, määrittely tulisi sijoittaa otsikkotiedostoon, joka sisältää luokkamallin määrittelyn.

Tämä on se muoto, jonka olemme valinneet käytettäväksi tässä kirjassa. Esimerkiksi mallien Queue ja QueueItem määrittelyt kuten myös niiden jäsenfunktioiden mallimäärittelyt ja staattiset tietojäsenet on sijoitettu otsikkotiedostoon Queue.h.

Kuten oli asianlaita funktiomallien määrittelyiden kanssa, liittyy luokkamallien jäsenten määrittelyiden sijoittamiseen otsikkotiedostoon joitakin huonoja puolia. Jäsenfunktioiden määrittelyt saattavat olla melko suuria ja ne voivat kuvata toteutuksen yksityiskohtia, joita käyttäjämme eivät välttämättä halua nähdä tai jotka haluamme piilottaa käyttäjiltämme. Lisäksi saman funktiomallin määrittelyiden kääntäminen moneen kertaan useissa eri tiedostoissa voi lisätä ohjelmiemme käännösaikaa tarpeettomasti. Jos erotteleva käännösmalli on käytettävissä, se mahdollistaa erottaa luokkamallin rajapinnan (tarkoittaa luokkamallin määrittelyä) sen toteutuksesta (tarkoittaa sen jäsenfunktioiden ja staattisten tietojäsenten määrittelyitä). Katsokaamme, kuinka voisimme käyttää sitä.

16.8.2 Erotteleva käännösmalli

Erottelevassa käännösmallissa luokkamallin määrittelyt ja sen välittömien jäsenfunktioiden määrittelyt sijoitetaan otsikkotiedostoon, kun taas muiden kuin välittömien jäsenfunktioiden ja staattisten jäsenfunktioiden määrittelyt sijoitetaan ohjelmatekstitiedostoon. Tässä käännösmallissa luokkamallin ja sen jäsenten määrittelyt järjestetään samalla tavalla kuin mallittomien luokkien ja niiden jäsenten määrittelyt. Esimerkiksi:

```
// ----- Queue.h -----
// esittelee Queue:n ulkoistetuksi luokkamalliksi
export template <class Type>
class Queue {
    // ...
public:
    Type& remove();
    void add( const Type & );
    // ....
};

// ----- Queue.C -----
// ulkoistettu Queue-luokkamallin määrittely otsikkotiedostossa Queue.h
#include "Queue.h"

template <class Type>
    void Queue<Type>::add( const Type &val ) { ... }

template <class Type>
    Type& Queue<Type>::remove() { ... }
```

Ohjelman, joka käyttää jäsenfunktion instantiointia, pitää ottaa mukaan otsikkotiedosto vain ennen instantioinnin käyttöä:

```
// ----- User.C -----
#include "Queue.h"

int main() {
    // instantioidaan Queue<int>
    Queue<int> *p_qi = new Queue<int>;
    int ival;
    // ...
    // ok: instantioidaan Queue<int>::add( const int & )
    p_qi->add( ival );
    // ...
}
```

Vaikka add()-jäsenfunktion mallimäärittely ei ole näkyvissä User.C-tiedostossa, voidaan malli-instantiointia Queue<int>::add(const int &) kutsua siitä huolimatta tässä tiedostossa. Jotta se olisi kuitenkin mahdollista, pitää luokkamalli esitellä erikoisella tavalla — *ulkoistettuna* (*exported*) luokkamallina.

Ulkoistettu luokkamalli on sellainen, josta vaaditaan vain luokkamallin määrittely jäsenfunktioiden tai staattisten tietojäsenten instantiointien käyttöön. Näiden jäsenten määrittelyt voidaan jättää pois tiedostoista, joissa instantiointeja käytetään.

Ulkoistettu luokkamalli esitellään laittamalla avainsana export ennen avainsanaa template luokkamallin määrittelyyn tai esittelyyn.

```
export template <class Type>
    class Queue { ... };
```

Esimerkissämme on käytetty avainsanaa export Queue-luokkamallille Queue.h-tiedostossa ja tämä otsikkotiedosto otetaan mukaan Queue.C-tiedostoon, joka sisältää luokkamallin jäsenfunktioiden määrittelyt. Jäsenfunktioiden add() ja remove() määrittelyt esitellään sitten automaattisesti ulkoistettuina. Näiden jäsenten määrittelyiden ei tarvitse olla mukana muissa tiedostoissa ennen kuin jäsenfunktioiden instantiointeja käytetään.

Huomaa, että vaikka luokkamalli on esitelty ulkoistetuksi, ei itse luokkamallin määrittelyä voi jättää pois User.C-tiedostosta. Queue<int>-luokan instantioinnille on User.C tiedostossa määrittely, jossa esitellään jäsenfunktiot Queue<int>::add() ja Queue<int>::remove(). Nämä esittelyt ovat tarpeen ennen kuin jäsenfunktioita kutsutaan. Joten, vaikka itse luokkamalli on esitelty ulkoistetuksi, export-avainsana vaikuttaa vain luokkamallin jäsenfunktioihin ja staattisiin tietojäseniin.

On myös mahdollista esitellä luokkamallin yksittäisiä jäseniä ulkoistetuiksi. Siinä tapauksessa export-avainsanaa ei määritetä itse luokkamalliin. Se määritetään vain tiettyihin ulkoistettaviin jäsenmäärittelyihin. Jos esimerkiksi Queue-luokkamallin kirjoittaja haluaa vain jäsenfunktion Queue<Type>::add() ulkoistettavaksi (haluaa vain poistaa tämän jäsenfunktion määrittelyn otsikkotiedostosta Queue.h), voidaan avainsana export määrittää vain add()-jäsenfunk-

tion määrittelyyn:

```
// ----- Queue.h -----
template <class Type>
class Queue {
    // ...
public:
    Type& remove();
    void add( const Type & );
};

// tarpeen, koska remove():a ei ole ulkoistettu
template <class Type>
    Type& Queue<Type>::remove() { ... }

// ----- Queue.C -----
#include "Queue.h"

// vain jäsenfunktio add() on ulkoistettu
export template <class Type>
    void Queue<Type>::add( const Type &val ) { ... }
```

Huomaa, että remove()-jäsenfunktion mallimäärittely on siirretty Queue.h-otsikkotiedostoon. Tämä on tarpeen, koska remove() ei enää ole ulkoistettu malli ja siten sen määrittelyn pitää olla näkyvissä tiedostoissa, joissa remove()-instantiointeja kutsutaan.

Luokkamallin jäsenfunktion tai staattisen tietojäsenen määrittely saa olla ulkoistettu vain kerran ohjelmassa. Koska kääntäjä prosessoi valitettavasti vain yhden tiedoston kerrallaan, se ei voi tavallisesti havaita, kun näitä jäseniä on määritelty ulkoisiksi useammassa kuin yhdessä ohjelmatiedostossa. Jos sellainen tilanne sattuu, tapahtuu jokin seuraavista:

1. Saadaan aikaan linkitysvirhe, joka ilmaisee, että samasta luokkamallin jäsenestä on tehty useampi kuin yksi mallimäärittely.
2. Kääntäjä voi instantioida jäsenen useammin kuin kerran samoilla malliargumenteilla saaden siten aikaan linkitysvirheen malli-instantioinnin tuplamäärittelyistä.
3. Toteutus voi instantioida jäsenen käyttäen yhtä ulkoistetuista mallimäärittelyistä ja jättää huomiotta muut määrittelyt.

Siksi ei ole varmaa, että virhe generoituu, jos ohjelmassamme on useampi kuin yksi luokkamallin jäsenen ulkoistettu määrittely. Meidän pitää olla huolellisia, kun järjestämme ohjelmiamme ja sijoitamme näitä jäsenmäärittelyitä vain yhteen ohjelmatekstitiedostoista.

Erotteleva käännösmalli mahdollistaa, että voimme erottaa kätevästi luokkamalliemme rajapinnat niiden toteutuksista ja voimme järjestää ohjelmamme niin, että luokkamalliemme rajapinta sijoitetaan otsikkotiedostoihin ja niiden toteutukset sijoitetaan tekstitiedostoihin. Kaikki kääntäjät eivät kuitenkaan tue erottelevaa käännösmallia tai jos tukevat, niin eivät kovin hyvin. Jos erottelevan käännösmallin tukea halutaan, tarvitaan hienostuneempia ohjelmointiympäristöjä eikä niitä ole saatavilla kaikista C++-toteutuksista.

Koska tämän kirjan mallien esimerkit ovat sangen pieniä ja koska haluamme esimerkkimme kääntyvän useimmissa C++-toteutuksissa, rajoitumme käyttämään mukaan ottavaa käännösmuotoa.

16.8.3 Eksplisiittisten instantiointien esittelyt

Kun käytetään mukaan ottavaa käännösmuotoa, otetaan luokkamallin jäsenmäärittely mukaan jokaiseen ohjelmatekstitiedostoon, joissa jotain sen instantioinneista käytetään. Ei tiedetä tarkkaan, missä ja milloin kääntäjä instantioi luokkamallin jäsenmäärittelyn ja jotkut kääntäjät (etenkin vanhemmat C++-kääntäjät) saattavat todella instantioida jäsenmäärittelyn tietyllä malliargumenttijoukolla useita kertoja. Näistä instantioinneista valitaan sitten yksi *siksi* instantioinniksi, jota ohjelma käyttää (kun ohjelma linkitetään tai jonkin esilinkitysvaiheen aikana). Muut instantioinnit yksinkertaisesti jätetään huomiotta.

Se, instantioidaanko jäsen kerran vai useita kertoja, ei vaikuta ohjelman tuloksiin, koska loppujen lopuksi ohjelma käyttää vain yhtä malli-instantiointia. Kuitenkin käännösaikaan vaikuttaa suuresti, jos malleja instantioidaan useita kertoja. Jos sovellus muodostuu lukuisista tiedostoista ja malli instantioidaan kaikissa näissä tiedostoissa, voi sovelluksen käännökseen kuluva aika kasvaa huomattavasti.

Aikaisempien kääntäjien instantiointiongelmat tekivät mallien käytöstä hankalan. Ratkaistakseen tämän C++-standardi otti käyttöön *eksplisiittiset instantiointiesittelyt*, joilla ohjelmoija sai kontrollin siitä, milloin instantiointi tapahtuu.

Eksplisiittinen instantiointiesittely on sellainen, jossa `template`-avainsanan jälkeen tulee `class`-avainsana ja luokkamallin instantioinnin nimi. Seuraavassa esimerkissä esitellään eksplisiittinen instantiointi `Queue<int>`-luokalle. Tämä eksplisiittinen instantiointi on pyyntö instantioida Queue-luokkamalli malliargumentilla `int`:

```
#include Queue.h

// eksplisiittinen instantiointiesittely
template class Queue<int>;
```

Kun luokkamalli instantioidaan eksplisiittisesti, kaikki sen jäsenet instantioidaan myös eksplisiittisesti samalla malliargumenttityypillä. Tämä merkitsee, että ei vain luokkamallin määrittelyä tarvitse laittaa tiedostoon, jossa eksplisiittinen instantiointiesittely esiintyy, vaan myös kaikki luokkamallin jäsenten määrittelyt pitää laittaa mukaan. Elleivät nämä määrittelyt ole mukana, eksplisiittinen instantiointiesittely on virheellinen. Esimerkiksi:

```
template <class Type>
    class Queue;

// virhe: Queue-mallia eikä sen jäseniä ole määritelty
template class Queue<int>;
```

Kun eksplisiittinen instantiointiesittely esiintyy ohjelmatekstitiedostossa, mitä tapahtuu

muissa tekstitiedostoissa, joissa luokkamallin instantiointia käytetään? Kuinka kerromme kääntäjälle, että eksplisiittinen instantiointiesittely esiintyy toisessa ohjelmatekstitiedostossa ja että luokkamallia ja sen jäseniä ei pidä instantioida, kun sitä käytetään ohjelman muissa tekstitiedostoissa?

Ratkaisu tässä tapauksessa on sama kuin se, jonka esitimme käsitellessämme funktiomalleja (kohdassa 10.5.3). Pitää käyttää kääntäjän valitsinta, joka vähentää mallien implisiittisiä instantiointeja. Kun käännämme sovelluksemme tällä valitsimella, kääntäjä olettaa, että käsittelemme malli-instantioinnit eksplisiittisillä instantiointiesittelyillä, jolloin se ei instantioi implisiittisesti malleja, joita sovelluksessamme käytetään.

Harjoitus 16.9

Minne sijoittaisit luokkamalliesi jäsenfunktioiden ja staattisten tietojäsenten määrittelyt, jos käyttämäsi toteutus tukee erottelevaa käännösmallia? Perustele, miksi.

Harjoitus 16.10

Käytä Screen-luokkamallia, jonka kehitit edellisten kohtien harjoituksissa (erityisesti jäsenfunktiot, jotka määrittelit kohdan 16.3 harjoituksessa 16.5 ja staattiset jäsenet, jotka määrittelit kohdan 16.5 harjoituksessa 16.7) ja järjestä nämä määrittelyt hyödyntämään luokkamallin erottelevaa käännösmallia.

16.9 Luokkamallin erikoistamiset ◆

Ennen kuin katsomme luokkamallin erikoistamisia ja näemme, miksi ohjelmiemme pitäisi määritellä niitä, lisätkäämme kaksi uutta jäsenfunktiota Queue-luokkamalliimme. Jäsenfunktiot min() ja max() iteroivat Queue-jäsenet läpi löytääkseen vastaavat minimi- ja maksimiarvot (Parempi olisi käyttää luvussa 12 esiteltyjä geneerisiä algoritmeja min() ja max(). Jotta kuitenkin voisimme esitellä mallien erikoistamisia, määrittelemme nämä funktiot Queue-luokkamallin jäsenfunktioiksi.) :

```
template <class Type>
class Queue {
    // ...
public:
    Type min();
    Type max();
    // ...
};


// etsi minimiarvo Queue:sta
template <class Type>
    Type Queue<Type>::min( )
{
```

```
    assert( ! is_empty() );
    Type min_val = front->item;
    for ( QueueItem *pq = front->next; pq != 0; pq = pq->next )
        if ( pq->item < min_val )
            min_val = pq->item;
    return min_val;
}

// etsi maksimiarvo Queue:sta
template <class Type>
    Type Queue<Type>::max( )
{
    assert( ! is_empty() );
    Type max_val = front->item;
    for ( QueueItem *pq = front->next; pq != 0; pq = pq->next )
        if ( pq->item > max_val )
            max_val = pq->item;
    return max_val;
}
```

Seuraava lause min()-jäsenfunktiossa vertailee kahta Queue-jäsentä:

```
pq->item < min_val
```

Tämä tuo esille piilovaatimuksen tyypeille, joilla Queue-luokkamalli on instantioitu: malliargumenttina käytetyn tyypin pitää kyetä käyttämään joko sisäisen tyypin esimääriteltyä pienempi kuin -operaattoria tai käyttäjän määrittelemää tyyppiä, jossa määritellään sen oma operator<()-operaattori. Ellei operator<()-operaattoria ole määritelty sellaiselle tyypille ja yritetään kutsua min()-funktiota tämän tyyppisille Queue-jäsenille, saadaan aikaan käännösvirhe siinä kohdassa, jossa kelpaamatonta vertailuoperaattoria on käytetty min()-funktiossa. (Samanlainen ongelma esiintyy max()-jäsenfunktion ja sen operator>()-operaattorin käytön kanssa.)

Olettakaamme, että meillä on seuraava tyyppi, jolla haluaisimme instantioida Queue-luokkamallin:

```
class LongDouble {
public:
    LongDouble( double dval ) : value( dval ) { }
    bool compareLess( const LongDouble & );
private:
    double value;
};
```

operator<()-operaattoria ei kuitenkaan ole olemassa kahden LongDouble-tyypin vertailuun eikä min()- ja max()-jäsenfunktioita voi käyttää Queue:n kanssa, joka on Queue<LongDouble>-tyyppi. Eräs ratkaisu tähän ongelmaan on määritellä globaalit operator<()- ja operator>()-operaattorit, jotka käyttävät LongDouble-tyypin compareLess()-jäsenfunktiota kahden Queue<LongDouble>-tyyppisen arvon vertailuun. Nämä globaalit operaattorit käynnistetään sitten automaattisesti min()- ja max()-funktioissa vertailemaan Queue<LongDouble>-tyyppisiä Queue-jäseniä. Esitelläk-

semme kuitenkin luokkamallien erikoistamisia, harkitsemme toista ratkaisua. Emme halua, että Queue-luokkamallin geneerisiä jäsenfunktioiden määrittelyitä käytetään min()- ja max()-jäsenfunktoiden instantiointiin, jos malliargumentti on luokkatyyppi LongDouble. Sen sijaan haluamme määritellä ilmentymät Queue<LongDouble>::min() ja Queue<LongDouble>::max(), jotka käyttävät LongDouble:n compareLess()-jäsenfunktiota.

Voimme tehdä tämän erikoistamalla luokkamallin jäsenen määrittelyn käyttäen *eksplisiittistä erikoistamismäärittelyä* (*explicit specialization definition*). Eksplisiittinen erikoistamismäärittely on sellainen, jossa template-avainsanan jälkeen tulee pienempi kuin (<)- ja suurempi kuin (>) -symbolit ja sen jälkeen luokkajäsenen erikoistamismäärittely. Seuraavassa esimerkissä on eksplisiittiset erikoistamiset määritelty luokkamallin Queue<LongDouble>-instantioinnin min()- ja max()-jäsenfunktioille:

```
// eksplisiittiset erikoistamismäärittelyt
template<> LongDouble Queue<LongDouble>::min( )
{
    assert( ! is_empty() );
    LongDouble min_val = front->item;
    for ( QueueItem *pq = front->next; pq != 0; pq = pq->next )
        if ( pq->item.compareLess( min_val ) )
            min_val = pq->item;
    return min_val;
}

template<> LongDouble Queue<LongDouble>::max( )
{
    assert( ! is_empty() );
    LongDouble max_val = front->item;
    for ( QueueItem *pq = front->next; pq != 0; pq = pq->next )
        if ( max_val.compareLess( pq->item ) )
            max_val = pq->item;
    return max_val;
}
```

Vaikka luokkatyyppi Queue<LongDouble> on instantioitu geneerisestä luokkamallin määrittelystä, jokainen Queue<LongDouble>-tyyppinen olio käyttää erikoistamisia min()- ja max()-jäsenfunktioille — näitä jäsenfunktioita ei instantioida Queue-luokkamallin geneerisistä jäsenfunktiomäärittelyistä.

Koska min()- ja max()-jäsenfunktioiden eksplisiittiset erikoistamismäärittelyt ovat funktiomäärittelyitä eivätkä mallimäärittelyitä, (ja koska näitä määrittelyitä ei ole esitelty välittömiksi) ei niitä voi sijoittaa otsikkotiedostoon. Ne pitää sijoittaa ohjelmatekstitiedostoon. Onneksi on mahdollista esitellä vain funktiomallin eksplisiittinen erikoistaminen määrittelemättä sitä. Esimerkiksi min()- ja max()-jäsenfunktioiden eksplisiittinen erikoistaminen voidaan esitellä seuraavasti:

```
// funktiomallin eksplisiittinen erikoistamisesittely
template<> LongDouble Queue<LongDouble>::min( );
template<> LongDouble Queue<LongDouble>::max( );
```

Kun sijoitamme nämä esittelyt otsikkotiedostoon ja näihin liittyvät määrittelyt ohjelmatekstitiedostoon, voimme siten järjestää koodin eksplisiittistä erikoistamista varten, kuten tekisimme minkä tahansa muun mallittoman luokan jäsenmäärittelyn yhteydessä.

Joissakin tapauksissa voi koko luokkamallimäärittely olla kelpaamaton jonkin tietyn tyypin käytölle. Siinä tapauksessa ohjelmoija voi tehdä määrittelyn erikoistaakseen koko luokkamallin. Ohjelmoija voi esimerkiksi tehdä täydellisen määrittelyn Queue<LongDouble>:sta:

```
// QueueLD.h: määrittelee luokkaerikoistamisen Queue<LongDouble>
#include "Queue.h"

template<> class Queue<LongDouble> {
    Queue<LongDouble>();
    ~Queue<LongDouble>();

    LongDouble& remove();
    void add( const LongDouble & );
    bool is_empty() const;
    LongDouble min();
    LongDouble max();
private:
    // Jokin tietty toteutus
};
```

Luokkamallin eksplisiittinen erikoistaminen voidaan määritellä vain sen jälkeen, kun yleinen luokkamalli on esitelty (vaikka yleistä mallia ei tarvitse määritellä). Tämä tarkoittaa, että nimen pitää olla tunnettu luokkamallin nimeksi ennen kuin malli voidaan erikoistaa. Jos edellisessä esimerkissä Queue.h-otsikkotiedostoa ei oteta mukaan ennen mallimäärittelyn eksplisiittistä erikoistamista, saadaan aikaan käännösvirhe, joka ilmaisee, että Queue ei ole mallin nimi.

Jos määrittelemme luokkamallin erikoistamisen, meidän pitää määritellä myös jokainen jäsenfunktio tai staattinen tietojäsen, joka liittyy tähän erikoistamiseen. Luokkamallin geneerisiä jäsenmäärittelyitä ei koskaan käytetä eksplisiittisen erikoistamisen jäsenmäärittelyiden luomiseen. Tämä johtuu siitä, että luokkamallin erikoistamisessa voi olla täysin erilaisia luokkajäseniä kuin geneerisessä mallissa. Jos päätämme tehdä eksplisiittisen erikoistamismäärittelyn luokkatyypille Queue<longDouble>, ei meidän pidä tehdä määrittelyitä ainoastaan min()- ja max()-jäsenfunktioille, vaan myös muiden jäsenfunktioiden määrittelyt yhtä lailla.

Jos koko luokka erikoistetaan, merkintä template<>, joka ilmaisee erikoistamismäärittelyä, sijoitetaan vain ennen luokkamallin eksplisiittistä erikoistamismäärittelyä. Luokkamallin erikoistamisen jäsenten määrittelyiden eteen ei saa laittaa merkintää template<>. Esimerkiksi:

```
#include "QueueLD.h"

// määrittelee luokkamallin erikoistamisen
// min()-jäsenfunktion
LongDouble Queue<LongDouble>::min( ) { }
```

Luokkamallia ei voi instantioida geneerisestä luokkamäärittelystä joissakin tiedostoissa ja erikoistaa toisissa tiedostoissa samoilla malliargumenteilla. Jos esimerkiksi malli QueueItem<LongDouble> erikoistetaan, pitää erikoistaminen esitellä jokaisessa tiedostossa, jossa sitä on käytetty:

```
// ---- File1.C ----
#include "Queue.h"

void ReadIn( Queue<LongDouble> *pq ) {
    // käytetään pq->add()
    // ja saadaan aikaan QueueItem<LongDouble>-instantiointi
}

// ---- File2.C ----
#include "QueueLD.h"

void ReadIn( Queue<LongDouble> * );

int main() {
    // käyttää erikoistamismäärittelyä QueueItem<LongDouble>
    Queue<LongDouble> *qld = new Queue<LongDouble>;

    ReadIn( qld );
    // ...
}
```

Edellinen ohjelma on virheellinen, vaikka toteutukset eivät useinkaan havaitse tällä tavalla virheellisiä ohjelmia. Tällaisten virheiden välttämiseksi QueueLD.h-otsikkotiedosto tulisi ottaa mukaan jokaiseen tiedostoon, jossa Queue<LongDouble>:a on käytetty ja ennen sen ensimmäistä käyttöä jokaisessa tiedostossa.

16.10 Luokkamallin osittainen erikoistaminen ◆A

Jos luokkamallilla on useampi kuin yksi malliparametri, voidaan luokkamalli haluttaessa erikoistaa tietylle malliargumentille tai -argumenteille erikoistamatta mallin kaikkia malliparametreja. Tämä tarkoittaa, että haluttaessa voidaan tehdä malli, joka vastaa geneeristä mallia, paitsi että jotkut malliparametreista on korvattu todellisilla tyypeillä tai arvoilla. Tämä on mahdollista, kun käytetään luokkamallin *osittaista erikoistamista* (*partial specialization*). Luokkamallin osittaista erikoistamista saatetaan tarvita, kun määritellään sopivampi tai tehokkaampi toteutus kuin geneerinen mallimäärittely tietylle malliargumenttijoukolle.

Käyttäkäämme esimerkiksi Screen-luokkamallia, joka esiteltiin kohdassa 16.2. Osittainen toteutus, Screen<hi,80>, saa aikaan tehokkaamman toteutuksen 80-sarakkeisille näytöille:

```
template <int hi, int wid>
class Screen {
    // ...
};

// Screen-luokkamallin osittainen erikoistaminen
template <int hi>
class Screen<hi, 80> {
public:
    Screen();
    // ...
private:
    string          _screen;
    string::size_type _cursor;
    short           _height;
    // Käyttää erityisalgoritmia 80-sarakkeisille näytöille
};
```

Luokkamallin osittainen erikoistaminen on malli ja osittaisen erikoistamisen määrittely näyttää mallimäärittelyltä. Sellainen määrittely alkaa template-avainsanalla, jonka jälkeen tulee malliparametriluettelo kulmasulkujen sisällä. Luokkamallin osittaisen erikoistamisen parametriluettelo eroaa vastaavasta geneerisen luokkamallin määrittelyn parametriluettelosta. Screen:in osittaisessa erikoistamisessa on vain yksi tyypitön malliparametri nimeltään hi, koska wid-malliargumentin tiedetään olevan 80. Osittaisen erikoistamisen malliparametriluettelossa luetellaan vain parametrit, joiden malliargumentit ovat yhä tuntemattomia.

Osittaisella erikoistamisella on sama nimi kuin geneerisellä mallilla, jota se vastaa; nimittäin Screen. Luokkamallin osittaisen erikoistamisen nimen jälkeen on kuitenkin aina malliargumenttiluettelo. Edellisessä esimerkissä malliargumenttiluettelo on <hi,80>. Koska ensimmäisen malliparametrin argumenttiarvo on tuntematon, argumenttiluettelo käyttää hi-malliparametrin nimeä paikanpitäjänä; toinen argumentti on arvo 80, jolle malli on osittain erikoistettu.

Luokkamallin osittainen erikoistaminen instantioidaan implisiittisesti, kun sitä käytetään ohjelmassa. Seuraavassa ohjelmassa luokkamallin osittainen erikoistaminen instantioidaan hi:n

malliargumentilla, joka on 24.

```
Screen<24,80> hp2621;
```

Huomaa, että instantiointi Screen<24,80> voidaan instantioida geneerisestä luokkamallin määrittelystä yhtä hyvin kuin osittaisesta erikoistamisesta. Miksi sitten osittainen erikoistaminen on valittu mallin instantiointiin? Kun luokkamallin osittaiset erikoistamiset on esitelty, kääntäjä valitsee mallimäärittelyn, joka on erikoistetuin instantiointia varten. Kun osittaista erikoistamista ei käytetä, käytetään geneerisen mallin määrittelyä. Kun esimerkiksi Screen<40,132> pitää instantioida, ei tämä instantiointi vastaa tehtyä osittaista erikoistamista. Osittaista erikoistamista käytetään vain, kun instantioidaan Screen-tyyppejä, joissa on 80 saraketta.

Osittaisen erikoistamisen määrittely on täydellisesti irrallaan geneerisen mallin määrittelystä. Osittaisessa erikoistamisessa saattaa olla täysin erilaiset jäsenet kuin geneerisessä luokkamallissa on. Luokkamallin osittaisella erikoistamisella pitää olla omat määrittelyt jäsenfunktioilleen, staattisille tietojäsenilleen ja sisäkkäisille tyypeilleen. Luokkamallin jäsenten geneerisiä määrittelyitä ei koskaan käytetä luokkamallin osittaisen erikoistamisen jäsenten instantiointiin. Esimerkiksi osittaiselle erikoistamiselle Screen<hi,80> pitää määritellä muodostaja. Tässä on mahdollinen määrittely:

```
// muodostaja osittaiselle erikoistamiselle Screen<hi,80>
template<int hi>
Screen<hi,80>::Screen() : _height( hi ), _cursor ( 0 ),
                          _screen( hi * 80, bk )
                        { }
```

Jos Screen<hi,80>:n muodostajan mallimäärittelyä ei tehdä ja osittaista erikoistamista on käytetty luokkatyypin instantiointiin, ei geneerisen luokkamallin muodostajamäärittelyä käytetä jäsenfunktion instantiointiin.

16.11 Nimiresoluutio luokkamalleissa ◆

Kun käsittelimme funktiomallien nimiresoluutiota kohdassa 10.9, mainitsimme, että nimiresoluutio etenee kahdessa vaiheessa. Nuo samat kaksi vaihetta pätevät myös sellaisten nimien resoluutioon, joita on käytetty luokkamallien ja niiden jäsenten määrittelyissä. Kumpikin vaihe pätee erilaisiin nimiin: ensimmäistä vaihetta käytetään nimiin, joilla on sama merkitys kaikissa luokkamallin instantioinneissa, ja toista vaihetta käytetään nimiin, joilla on todennäköisesti eri merkitys instantioinnista toiseen. Katsotaanpa joitakin esimerkkejä Queue-luokkamallin remove()-jäsenfunktion käytöstä:

```
// Queue.h:
#include <iostream>
#include <cstdlib>

// Queue-luokan määrittely
```

```
template <class Type>
Type Queue<Type>::remove() {
    if ( is_empty() ) {
        cerr << "remove() on empty queue\n";
        exit( -1 );
    }

    QueueItem<Type> *pt = front;
    front = front->next;
    Type retval = pt->item;
    delete pt;

    cout << "value removed: ";
    cout << retval << endl;

    return retval;
}
```

Lausekkeessa

```
cout << retval << endl;
```

retval on Type-tyyppi eikä sen todellista tyyppiä tiedetä ennen kuin jäsenfunktio remove() on instantioitu. Valittu operator<<()-operaattori riippuu retval:in todellisesta tyypistä, joka tarkoittaa tyyppiä, jolla malliparametri Type korvataan. Siksi on mahdotonta tietää, mitä operator<<()-operaattoria kutsutaan, kunnes remove() instantioidaan. remove():n erilaiset instantioinnit kutsuvat todennäköisesti eri operator<<()-operaattoria. Tämän vuoksi sanomme, että valittu operator<<()-operaattori *riippuu* malliparametrista.

Tilanne on kuitenkin erilainen, kun kutsutaan exit()-funktiota. Kun kutsutaan exit()-funktiota, on funktion argumentti literaali, jonka arvo on sama kaikissa remove()-jäsenfunktion instantioinneissa. Koska funktion kutsu ei käytä argumenttityyppejä, jotka riippuvat malliparametrista Type, on taattu, että exit()-funktiota kutsuttaessa kaikki instantioinnit käynnistävät exit()-funktion, joka on esitelty cstdlib-otsikkotiedostossa. Samalla tavalla tiedämme, että lausekkeen

```
cout << "value removed: ";
```

globaalia operaattoria

```
ostream& operator<<( ostream &, const char * );
```

aina kutsutaan. Argumentti "value removed: " on C-tyylinen merkkijonotyyppi, joka ei riipu Type-malliparametrista. On siksi taattu, että tällä operator<<()-operaattorin käytöllä on sama merkitys kaikissa remove()-jäsenfunktion instantioinneissa. Rakenne, jolla on sama merkitys mallin kaikissa instantioinneissa, on sellainen, joka *ei* riipu malliparametrista.

Luokkamallien tai sen jäsenten määrittelyiden nimiresoluution kaksi vaihetta ovat tästä syystä seuraavat:

1. Nimet, jotka eivät riipu malliparametrista, ratkaistaan, kun malli määritellään.
2. Nimet, jotka riippuvat malliparametrista, ratkaistaan, kun malli instantioidaan.

Tämä kaksivaiheinen lähestymistapa tukee sekä luokkamallin suunnittelijan että käyttäjän vaatimuksia. Luokkamallin suunnittelijana haluamme kontrolloida niin paljon kuin mahdollista, kuinka mallimäärittelyn nimet ratkaistaan. Jos luokkamalli on osa kirjastoa, jossa on määritelty myös muita malleja ja funktioita, haluamme luokkamallin instantiointien ja sen jäsenten käyttävän kirjastomme muita komponentteja aina, kun se on mahdollista. Nimiresoluution ensimmäinen vaihe takaa, että tämä tapahtuu. Kun mallimäärittelyssä käytetty nimi ei riipu malliparametrista, ratkaistaan se ottamalla huomioon vain esittelyt, jotka näkyvät otsikkotiedostossa ennen mallimäärittelyä.

Itse asiassa luokkamallin suunnittelijan pitää varmistua, että kaikille mallimäärittelyssä käytetyille nimille, jotka eivät riipu malliparametrista, tehdään esittely. Jos mallimäärittelyssä käytetty nimi ei riipu malliparametrista eikä tämän nimen esittelyä löydy, kun malli määritellään, mallimäärittely on virheellinen. Ellei otsikkotiedostoja iostream ja cstdlib olisi otettu mukaan ennen Queue-luokkamallin remove()-jäsenfunktion määrittelyä, olisi lauseke

```
cout << "value removed: ";
```

tai exit()-kutsu ollut virheellinen.

Nimiresoluution toinen vaihe on tarpeellinen, jos funktiot ja operaattorit, jotka liittyvät tyyppiin, jolla malli on instantioitu, otetaan huomioon. Jos esimerkiksi Queue-luokkamalli instantioidaan kohdassa 16.9 määritellyllä LongDouble-luokkatyypillä, haluamme seuraavan lausekkeen Queue:n remove()-jäsenfunktiossa

```
cout << retval << endl;
```

käynnistävän LongDouble-luokkaan liittyvän operator<<()-tulostusoperaattorin. Esimerkiksi:

```
#include "Queue.h"
#include "ldouble.h"
// sisältää:
// class LongDouble { ... };
// ostream& operator<<( ostream &, const LongDouble & );

int main() {
    // instantioidaan Queue<LongDouble>
    Queue<LongDouble> *qld = new Queue<LongDouble>;

    // instantioidaan Queue<LongDouble>::remove()
    // käynnistää tulostusoperaattorin LongDouble-luokkatyypille
    qld->remove();
    //...
}
```

Täsmällistä kohtaa, jossa malli instantioidaan, kutsutaan mallin *instantiointikohdaksi*. Sen tietäminen, missä kohdassa mallin instantiointi sijaitsee, on tärkeää, koska se määrää, mitkä esittelyt otetaan huomioon nimille, jotka riippuvat malliparametrista.

Luokkamallin instantiointikohta on aina nimiavaruuden viittausalueella ja se on aina välittömästi ennen esittelyä tai määrittelyä, joka viittaa luokkamallin instantiointiin. Luokkamallin jäsenfunktion tai staattisen tietojäsenen instantiointikohta on aina välittömästi esittelyn tai määrittelyn jälkeen, joka viittaa luokkamallin jäsenen instantiointiin.

Edellisessä esimerkissä Queue<LongDouble>:n instantiointikohta edeltää välittömästi main()-funktiota ja kääntäjä ottaa huomioon kaikki esittelyt ennen tätä kohtaa ratkaistakseen malliparametrista riippuvat nimet, joita käytetään Queue-mallimäärittelyssä. remove()-jäsenfunktion instantiointikohta on välittömästi main()-funktion jälkeen ja kääntäjä ottaa huomioon kaikki esittelyt ennen tätä kohtaa ratkaistakseen malliparametrista riippuvat nimet, joita käytetään remove()-jäsenfunktion määrittelyssä.

Kuten mainitsimme kohdassa 16.2, luokkamalli instantioidaan, jos sitä käytetään yhteydessä, joka vaatii luokan täydellisen määrittelyn. Luokkamallin instantioinnin jäseniä ei instantioida automaattisesti, kun luokkamalli instantioidaan. Sen sijaan jäsenet instantioidaan vain, jos niitä käytetään ohjelmassa. Tästä syystä luokkamallin instantiointikohta voi olla eri kuin sen jäsenten instantiointikohta ja eri jäsenillä voi olla eri instantiointikohdat. Virheiden välttämiseksi luokkamallin ja sen jäsenten määrittelyissä käytettyjen nimien esittelyt tulisi sijoittaa otsikkotiedostoihin ja ottaa mukaan ennen luokkamallin ensimmäistä instantiointia ja ennen yhdenkään sen jäsenen instantiointia.

16.12 Nimiavaruudet ja luokkamallit ◆

Luokkamallin määrittely voidaan sijoittaa nimiavaruuteen kuten mikä tahansa muu globaalin viittausalueen määrittely. (Katso kohdista 8.5 ja 8.6 nimiavaruuksien käsittely.) Sellaisen mallimäärittelyn tarkoitus on sama kuin mallilla, joka on määritelty globaalille viittausalueelle paitsi, että mallin nimi on piilotettu nimiavaruuteen. Mallin nimi pitää tarkentaa nimiavaruuden nimellä, kun mallia käytetään nimiavaruutensa ulkopuolella tai sitten pitää käyttää using-esittelyä. Esimerkiksi:

```
#include <iostream>
#include <cstdlib>

namespace cplusplus_primer {

    template <class Type>
    class Queue { // ...
    };

    template <class Type>
    Type Queue<Type>::remove()
    {
```

```
        // ...
    }
}
```

Luokkamallin Queue-nimi pitää tarkentaa cplusplus_primer-nimiavaruuden nimellä tai käyttää using-esittelyä. Muutoin Queue-luokkamallia käytetään kuten tässä luvussa on aikaisemmin kuvattu — se instantioidaan samalla tavalla ja sillä voi olla jäsenfunktioita, staattisia tietojäseniä ja sisäkkäisiä tyyppejä jne. Esimerkiksi:

```
int main() {
    using cplusplus_primer Queue; // using-esittely

    // viittaa cplusplus_primer-nimiavaruuden luokkamalliin
    Queue<int> *p_qi = new Queue<int>;
    // ...
    p_qi->remove();
}
```

Malli cplusplus_primer::Queue instantioidaan, koska sitä on käytetty new-lausekkeessa

```
... = new Queue<int>;
```

p_qi on osoitin luokkatyyppiin cplusplus_primer::Queue<int>. Kun tätä osoitinta käytetään remove()-jäsenfunktioon viittaamiseen, se viittaa tämän malli-instantioinnin remove()-jäsenfunktioon.

Kun luokkamalli esitellään nimiavaruudessa, se vaikuttaa myös siihen, kuinka luokkamallin ja sen jäsenten erikoistamisia ja osittaisia erikoistamisia esitellään. (Erikoistamiset on käsitelty kohdassa 16.9 ja osittainen erikoistaminen kohdassa 16.10). Luokkamallin tai sen jäsenen erikoistamisesittely pitää esiintyä siinä nimiavaruudessa, jossa geneerinen malli on määritelty.

Seuraavassa esimerkissä on luokkatyyppien Queue-<char*> ja Queue<double> jäsenfunktion remove() erikoistamisesittelyt tehty cplusplus_primer-nimiavaruuteen.

```
#include <iostream>
#include <cstdlib>

namespace cplusplus_primer {

    template <class Type>
    class Queue { ... };

    template <class Type>
    Type Queue<Type>::remove() { ... }

    // erikoistamisesittely:
    // cplusplus_primer::Queue<char*>
    template<> class Queue<char*> { ... };

    // erikoistamisesittely:
    // cplusplus_primer::Queue<double>::remove()-jäsenfunktiolle
    template<> double Queue<double>::remove() { ... }
```

```
}
```

Vaikka erikoistamiset ovat cplusplus_primer-nimiavaruuden jäseniä, ei niiden määrittelyiden tarvitse esiintyä cplusplus_primer-nimiavaruudessa sinänsä. On mahdollista määritellä mallin erikoistaminen nimiavaruutensa ulkopuolelle edellyttäen, että määrittely esiintyy nimiavaruudessa, joka ympäröi cplusplus_primer-nimiavaruutta ja että erikoistamisen nimi on asianmukaisesti tarkennettu nimiavaruuden nimellä. Esimerkiksi:

```
namespace cplusplus_primer
{
    // Queue:n ja sen jäsenfunktioiden määrittely

}

// erikoistamisesittely:
// cplusplus_primer::Queue<char*>
template<> class cplusplus_primer::Queue<char*> { ... };

// erikoistamisesittely jäsenfunktiolle
// cplusplus_primer::Queue<double>::remove()
template<> double cplusplus_primer::Queue<double>::remove()
    { ... }
```

Erikoistamisesittelyt `cplusplus_primer::Queue<char*>`-luokkatyypille ja `cplusplus_primer::Queue<double>`-luokkatyyppiselle `remove()`-jäsenfunktiolle on tehty globaalille viittausalueelle. Koska globaali viittausalue ympäröi cplusplus_primer-nimiavaruutta ja koska erikoistamisien nimet on tarkennettu cplusplus_primer-nimiavaruuden nimellä, ovat nämä määrittelyt kelvollisia Queue-luokkamallin erikoistamisia cplusplus_primer-nimiavaruudessa.

16.13 Array-luokkamalli

Tässä kohtaa viemme loppuun kohdassa 2.5 esitellyn Array-luokkamallin toteutuksen (tätä luokkamallia on laajennettu yhden periytymisen kautta kohdassa 18.3 ja moniperiytymisen kautta kohdassa 18.6). Tässä on täydellinen otsikkotiedosto Array-luokkamallille:

```
#ifndef ARRAY_H
#define ARRAY_H
#include <iostream>

template <class elemType> class Array;
template <class elemType> ostream&
    operator<<( ostream &, const Array<elemType> & );

template <class elemType>
class Array {
public:
    explicit Array( int sz = DefaultArraySize )
        { init( 0, sz ); }
```

```
    Array( const elemType *ar, int sz )
        { init( ar, sz ); }
    Array( const Array &iA )
        { init( iA._ia, iA._size ); }
    ~Array() { delete[] _ia; }

    Array & operator=( const Array & );
    int size() const { return _size; }

    elemType& operator[]( int ix ) const
        { return _ia[ix]; }

    ostream &print( ostream &os = cout ) const;
    void grow();

    void sort( int,int );
    int find( elemType );
    elemType min();
    elemType max();
private:
    void init( const elemType *, int );
    void swap( int, int );

    static const int DefaultArraySize = 12;

    int _size;
    elemType *_ia;
};

#endif
```

Kolmen muodostajan toteutuksen yhteinen koodi on sijoitettu riippumattomaan jäsenfunktioon nimeltään `init()`. Koska ei ole aikomus, että Array-luokkamallin käyttäjät käynnistävät sen, siitä on tehty yksityinen jäsen.

```
template <class elemType>
    void Array<elemType>::init( const elemType *array, int sz )
{
    _size = sz;
    _ia = new elemType[ _size ];

    for ( int ix = 0; ix < _size; ++ix )
        if ( ! array )
            _ia[ ix ] = 0;
        else _ia[ ix ] = array[ ix ];
}
```

Kopioinnin sijoitusoperaattorin toteutus on selkeä. Kuten mainitsimme kohdassa 14.7, tämän operaattorin toteutus vartioi sitä, ettei olioita kopioida itseensä:

```
template <class elemType> Array<elemType>&
    Array<elemType>::operator=( const Array<elemType> &iA )
{
    if ( this != &iA ) {
        delete[] _ia;
        init( iA._ia, iA._size );
    }
    return *this;
}
```

Jäsenfunktio print() käsittelee olion varsinaisen tulostuksen tyypillä, joka on Array-luokkamallin instantiointi. Sen tulostus on huolitellumpi kuin ehkä on tarpeen, mutta se tulostaa hienosti. Jos Array<int>-instantiointi sisältää elementit 3, 5, 8, 13 ja 21, olion tulostus näyttää tältä:

```
(5) < 3, 5, 8, 13, 21 >
```

Tulostusoperaattori ostream käynnistää yksinkertaisesti print()-funktion. Tässä on molempien funktioiden toteutukset:

```
template <class elemType> ostream&
    operator<<( ostream &os, const Array<elemType> &ar )
{
    return ar.print( os );
}

template <class elemType>
    ostream & Array<elemType>::print( ostream &os ) const
{
    const int lineLength = 12;

    os << "( " << _size << " )< ";
    for ( int ix = 0; ix < _size; ++ix )
    {
        if ( ix % lineLength == 0 && ix )
            os << "\n\t";
        os << _ia[ ix ];

        // älä generoi pilkkua rivin viimeisen jäsenen jälkeen
        // äläkä taulukon viimeisen elementin jälkeen
        if ( ix % lineLength != lineLength-1 &&
             ix != _size-1 )
            os << ", ";
    }
    os << " >\n";
    return os;
}
```

print()-jäsenfunktiossa oleva lause, joka käsittelee Array-elementin arvon varsinaisen tulostuksen

```
os << _ia[ ix ];
```

tuo esille piilovaatimuksen tyypeille, joilla Array-luokkamallia instantioidaan: malliargumenttina käytetyn tyypin pitää olla joko sisäinen tyyppi tai käyttäjän määrittelemä luokkatyyppi, jossa on määriteltynä oma tulostusoperaattori. Jos tulostusoperaattoria ei ole määritelty sellaiselle tyypille ja tällä tyypillä instantioidun Array:n sisältö yritetään näyttää, saadaan siitä aikaan kääntäjän virheilmoitus kohdassa, jossa kelpaamatonta tulostusoperaattoria yritettiin käyttää.

grow()-jäsenfunktio kasvattaa Array-olion kokoa. Esimerkissämme se kasvattaa Array-oliota yksinkertaisesti puolella sen nykyisestä koosta:

```
template <class elemType>
    void Array<elemType>::grow()
{
    elemType *oldia = _ia;
    int oldSize = _size;

    _size = oldSize + oldSize/2 + 1;
    _ia = new elemType[_size];

    int ix;
    for ( ix = 0; ix < oldSize; ++ix)
        _ia[ix] = oldia[ix];

    for ( ; ix < _size ++ix )
      _ia[ix] = elemType();
    delete[] oldia;
}
```

Jäsenfunktiot find(), min() ja max() toteuttavat iteratiivisen etsinnän läpi sisäisen _ia-tulukon. Nämä jäsenfunktiot voitaisiin toteuttaa paljon tehokkaammin, tietysti olettaen, että taulukko on varmasti lajiteltu.

```
template <class elemType>
    elemType Array<elemType>::min( )
{
    assert( _ia != 0 );
    elemType min_val = _ia[0];
    for ( int ix = 1; ix < _size; ++ix )
        if ( _ia[ix] < min_val )
            min_val = _ia[ix];
    return min_val;
}

template <class elemType>
    elemType Array<elemType>::max()
```

```
{
    assert( _ia != 0 );
    elemType max_val = _ia[0];
    for ( int ix = 1; ix < _size; ++ix )
        if ( max_val < _ia[ix] )
            max_val = _ia[ix];
    return max_val;
}

template <class elemType>
    int Array<elemType>::find( elemType val )
{
    for ( int ix = 0; ix < _size; ++ix )
        if ( val == _ia[ix] ) return ix;
    return -1;
}
```

Array-luokkamallissa on lopussa sort()-jäsenfunktio. Sen toteutuksessa on käytetty quicksort-algoritmia. Jäsenfunktio näyttää melko samanlaiselta kuin funktiomallin toteutus, joka määriteltiin kohdassa 10.11. Tarkoitus on, että swap() toimii yksinkertaisesti apufunktiona sort()-jäsenfunktiolle. Se ei ole osa Array-luokkamallin julkista rajapintaa ja on siksi tehty yksityiseksi jäseneksi:

```
template <class elemType>
    void Array<elemType>::swap( int i, int j )
{
    elemType tmp = _ia[i];
    _ia[i] = _ia[j];
    _ia[j] = tmp;
}

template <class elemType>
    void Array<elemType>::sort( int low, int high )
{
    if ( low >= high ) return;
    int lo = low;
    int hi = high + 1;
    elemType elem = _ia[low];

    for (;;) {
        while ( _ia[++lo] < elem && lo < high ) ;
        while ( _ia[--hi] > elem && hi > low ) ;
        if ( lo < hi )
            swap( lo, hi );
        else break;
    }

    swap( low, hi );
    sort( low, hi-1 );
```

```
    sort( hi+1, high );
}
```

Koodin toteuttaminen ei tietenkään takaa, että koodi todella toimii. try_array() on mallifunktio, jonka tarkoitus on testata Array-luokkamallin toteutusta. Se näyttää tältä:

```
#include "Array.h"

template <class elemType>
    void try_array( Array<elemType> &iA )
{
    cout << "try_array: initial array values:\n";
    cout << iA << endl;

    elemType find_val = iA [ iA.size()-1 ];
    iA[ iA.size()-1 ] = iA.min();

    int mid = iA.size()/2;
    iA[0] = iA.max();
    iA[mid] = iA[0];
    cout << "try_array: after assignments:\n";
    cout << iA << endl;

    Array<elemType> iA2 = iA;
    iA2[mid/2] = iA2[mid];
    cout << "try_array: memberwise initialization\n";
    cout << iA << endl;

    iA = iA2;
    cout << "try_array: after memberwise copy\n";
    cout << iA << endl;

    iA.grow();
    cout << "try_array: after grow\n";
    cout << iA << endl;

    int index = iA.find( find_val );
    cout << "value to find: " << find_val;
    cout << "\tindex returned: " << index << endl;

    elemType value = iA[index];
    cout << "value found at index: ";
    cout << value << endl;
}
```

Katsotaanpa try_array()-funktiomalliamme. Ensimmäinen vaihe on tulostaa alkuperäinen Array. Tämä vahvistaa mallin tulostusoperaattorin instantioinnin ja antaa meille pikakuvan alkuperäisestä Array:sta, johon voimme verrata tulevia onnistuneita (tai epäonnistuneita) muokkauksia. find_val sisältää arvon, joka myöhemmin välitetään find()-funktiolle. Jos try_array()

olisi malliton funktio, olisi arvo ollut literaalivakio. Koska kuitenkaan yksikään arvo ei toimi tyyppinä jokaisessa mahdollisessa instantioinnissa, ei arvo voi olla literaalivakio. Array:n elementit sijoitetaan satunnaisesti Array:n toisiin elementteihin tutkimalla min()-, max()-, size()-funktioita ja tietysti indeksioperaattoria.

iA2 on alustettu jäsenittäin iA:lla käynnistämällä Array-luokkamallin kopiointimuodostaja. iA2 kokeilee sitten indeksioperaattoriaan sijoittamalla elementtiin mid/2. (Nämä kaksi riviä ovat vielä kiinnostavampia, kun iA on todella Array:sta johdettu alityyppi ja indeksioperaattori on esitelty virtuaalifunktiona. Katsomme tätä jälleen luvussa 18, kun käsittelemme periytymistä.) iA kopioidaan seuraavaksi jäsenittäin muokatulla iA2:lla käynnistämällä Array-luokan sijoitusoperaattori. Sen jälkeen sekä grow()- että find()-jäsenfunktiot tutkitaan. Funktio jättää tahallaan tutkimatta find()-funktion paluuarvon. Muista, että find() palauttaa arvon -1, ellei sen etsimää elementtiä löydy. Jos Array-indeksin arvo on -1, se saa aikaan alivuotovirheen. (Luvussa 18 johdetaan Array:sta raja-arvot tutkiva Array-luokkamalli, joka sieppaa tämän virheen.)

Haluaisimme varmistua, että mallin toteutuksemme toimii monilla tietotyypeillä — esimerkiksi kokonaisluvuilla, liukuluvuilla ja merkkijonoilla. Tässä on versio main()-funktiosta, jossa try_array() tutkitaan jokaisella näillä kolmella tietotyypillä:

```
#include "Array.C"
#include "try_array.C"
#include <string>

int main()
{
    static int ia[] = { 12,7,14,9,128,17,6,3,27,5 };
    static double da[] = {12.3,7.9,14.6,9.8,128.0 };
    static string sa[] = { "Eeyore", "Pooh", "Tigger",
                    "Piglet", "Owl", "Gopher", "Heffalump" };

    Array<int> iA( ia, sizeof(ia)/sizeof(int) );
    Array<double> dA( da, sizeof(da)/sizeof(double) );
    Array<string> sA( sa, sizeof(sa)/sizeof(string) );

    cout << "template Array<int> class\n" << endl;
    try_array(iA);

    cout << "template Array<double> class\n" << endl;
    try_array(dA);

    cout << "template Array<string> class\n" << endl;
    try_array(sA);

    return 0;
}
```

Seuraavassa on tulostus, kun Array-luokkamallin instantiointi on double-tyyppinen:

```
try_array: initial array values:
( 5 )< 12.3, 7.9, 14.6, 9.8, 128 >
```

```
try_array: after assignments:
( 5 )< 14.6, 7.9, 14.6, 9.8, 7.9 >

try_array: memberwise initialization
( 5 )< 14.6, 7.9, 14.6, 9.8, 7.9 >

try_array: after memberwise copy
( 5 )< 14.6, 14.6, 14.6, 9.8, 7.9 >

try_array: after grow
( 8 )< 14.6, 14.6, 14.6, 9.8, 7.9, 0
    0, 0 >

value to find: 128index returned: -1
value found at index: 3.35965e-322
```

Yli raja-arvojen menevä indeksi saa aikaan sen, että ohjelman viimeksi palauttama arvo on kelpaamaton. Sama virhetilanne saa aikaan Array-luokkamallin string-instantioinnin romahtamisen eli “kaatumisen” suorituksen aikana. Tässä on tuo tulostus:

```
template Array<String> class

try_array: initial array values:
( 7 )< Eeyore, Pooh, Tigger, Piglet, Owl, Gopher
    Heffalump >

try_array: after assignments:
( 7 )< Tigger, Pooh, Tigger, Tigger, Owl, Gopher
    Eeyore >

try_array: memberwise initialization
( 7 )< Tigger, Pooh, Tigger, Tigger, Owl, Gopher
    Eeyore >

try_array: after memberwise copy
( 7 )< Tigger, Tigger, Tigger, Tigger, Owl, Gopher
    Eeyore >

try_array: after grow
( 11 )< Tigger, Tigger, Tigger, Tigger, Owl, Gopher
    Eeyore, <empty>, <empty>, <empty>, <empty> >

value to find: Heffalumpindex returned: -1
Memory fault(coredump)
```

Harjoitus 16.11

Muuta tässä kohdassa määriteltyä Array-luokkamallia niin, että poistat siitä jäsenfunktiot sort(), find(), max(), min() ja swap() sekä muutat try_array()-funktiomallin käyttämään niiden sijaan geneerisiä algoritmeja (määritelty luvussa 12) .