10

Funktiomallit

Tässä luvussa kuvataan, mikä funktiomalli on, ja tuodaan esille, kuinka niitä määritellään ja käytetään. Funktiomallin käyttö on melko yksinkertaista ja monet aloittelevat C++-ohjelmoijat käyttävät kirjastoihin määriteltyjä funktiomalleja tietämättä, että he käyttävät niitä. Vain pidemmälle ehtineet C++-käyttäjät määrittelevät ja käyttävät funktiomalleja kuten tässä luvussa kerrotaan. Siten tämän luvun materiaali toimii johdatusmateriaalina C++:n edistyneemmille aihealueille. Aloitamme tämän luvun kuvaamalla, mikä funktiomalli on ja kuinka sellainen määritellään. Sitten kuvaamme funktiomallien yksinkertaisia käyttötapoja. Sitten luvussa siirrytään edistyneempiin aiheisiin. Aluksi katsomme, kuinka funktiomallia käytetään edistyneemmillä tavoilla: katsomme malliargumentin päättelyä yksityiskohtaisesti ja kuinka eksplisiittiset malliargumentit voidaan määrittää, kun viitataan funktiomallin ilmentymään. Katsomme sen jälkeen, kuinka kääntäjä instantioi (tekee ilmentymiä) malleja ja selvitämme vaatimukset, joita tämä edellyttää ohjelmiemme rakenteelta sekä käsittelemme, kuinka määrittelemme erityisominaisuuksia funktiomallin ilmentymälle. Sitten luvussa käsitellään aiheita, jotka kiinnostavat funktiomallien suunnittelijoita. Kerromme, kuinka funktiomalleja voidaan ylikuormittaa ja kuinka funktiomallien ylikuormituksen ratkaisu toimii. Katsomme myös funktiomallien määrittelyjen nimiresoluutiota ja kerromme, kuinka funktiomalleja voidaan määritellä nimiavaruuksiin. Luku päättyy esimerkkiin funktiomallin käytöstä.

10.1 Funktiomallin määrittely

Vahvasti tyypitetty kieli voi joskus vaikuttaa esteeltä toteuttaa jotain sellaista, joka muutoin on selkeää funktioille. Vaikka esimerkiksi seuraavan `min()`-funktion algoritmi vaikuttaa selkeältä, vahva tyypitys vaatii, että toteutamme ilmentymän molemmille tyyppipareille, joita haluamme verrata:

```
int min( int a, int b ) {
    return a < b ? a : b;
}
```

```
double min( double a, double b ) {
    return a < b ? a : b;
}
```

Houkutteleva, mutta hieman vaarallinen vaihtoehto on käyttää esikääntäjän makrolaajennusta jokaisen min()-funktion ilmentymän eksplisiittiselle määrittelylle. Esimerkiksi:

```
#define min(a,b) ((a) < (b) ? (a) : (b))
```

Vaikka tämä määrittely toimii oikein min()-funktion yksinkertaisille kutsuille kuten

```
min( 10, 20 );
min( 10.0, 20.0 );
```

se toimii odottamattomalla tavalla monimutkaisissa kutsuissa. Se johtuu siitä, että sen mekanismi ei käyttäydy kuten funktion kutsu, vaan muodostaa korvaavan tekstin argumenteistaan. Tuloksena on, että sen kahden argumentin arvot arvioidaan *kahdesti*: ensimmäisen kerran silloin, kun argumentteja a ja b testataan ja toisen kerran silloin, kun makron palauttamaa arvoa tutkitaan. Esimerkiksi:

```
#include <iostream>
#define min(a,b) ((a) < (b) ? (a) : (b))

const int size = 10;
int ia[size];

int main() {
    int elem_cnt = 0;
    int *p = &ia[0];

    // laske taulukon elementtien lukumäärä
    while ( min(p++,&ia[size]) != &ia[size] )
        ++elem_cnt;

    cout << "elem_cnt : "   << elem_cnt
       << "\texpecting: " << size << endl;
    return 0;
}
```

Myönnettäköön, että tässä ohjelmassa on käytetty kiertotietä ia-kokonaislukutaulukon elementtien laskemiseen. Makron min() laajennus epäonnistuu tässä tapauksessa, koska sen p-osoitinargumenttiin tarkoitettua jälkiliitekasvatusoperaatiota käytetään kahdesti joka laajennuksessa. Tämän ohjelman suorituksen tuloksena syntyy seuraava virheellinen laskutulos:

```
elem_cnt : 5    expecting: 10
```

Funktiomallit tarjoavat mekanismin, joilla voimme säilyttää funktioiden määrittelyjen ja funktioiden kutsujen merkityksen (kapseloida koodikatkelman yhteen ohjelmapaikkaan ja varmistua, että argumentit arvioidaan vain kerran ennen funktion käynnistystä) ilman, että pitää ohittaa C++:n vahva tyyppitarkistus, kuten tehdään makroratkaisussa.

Funktiomallissa on algoritmi, jota käytetään automaattisesti generoitaessa tiettyjä ilmentymiä funktiosta vaihtelevin tyypein. Ohjelmoija *parametroi* kaikki tai osan funktion rajapinnan tyypeistä (parametri- ja paluutyypit), jolloin funktion runko muutoin jää muuttumattomaksi. Funktio on ehdolla tehtäväksi malliksi silloin, kun sen toteutus säilyy muuttumattomana ilmentymissä, joista jokainen käsittelee samanlaisia tietotyyppejä kuin min()-funktiomme.

Tässä on esimerkki min()-funktiomallin määrittelystä:

```
template <class Type>
    Type min( Type a, Type b ) {
        return a < b ? a : b;
}

int main() {
    // ok: int min( int, int );
    min( 10, 20 );

    // ok: double min( double, double );
    min( 10.0, 20.0 );
    return 0;
}
```

Jos korvaamme funktiomallilla esikääntäjän min()-makron edellisestä ohjelmasta, on ohjelman tulostus nyt laskettu oikein:

```
elem_cnt : 10    expecting: 10
```

(C++-vakiokirjastossa on funktiomalleja yleisimmin käytetyille algoritmeille kuten tässä määritellylle min()-funktiolle. Nämä algoritmit kuvataan luvussa 12. Jotta voisimme esitellä funktiomalleja, määrittelemme omia yksinkertaistettuja versioita joistakin algoritmeista, jotka on määritelty C++-vakiokirjastoon.)

Avainsana template aloittaa aina sekä määrittelyn että esittelyn funktiomallista. Avainsanan jälkeen tulee pilkuin eroteltu luettelo malliparametreja, jotka ovat kulmasulkujen sisällä (< >). Tämä luettelo on *malliparametriluettelo*. Se ei voi olla tyhjä. Malliparametri voi olla *mallityyppiparametri*, joka edustaa tyyppiä, tai *mallityypitön parametri*, joka edustaa vakiolauseketta.

Mallityyppiparametri muodostuu avainsanasta class tai typename, jonka jälkeen tulee tunnus. Funktion malliparametriluettelossa näiden avainsanojen merkitys on sama. Ne ilmaisevat, että seuraava parametrinimi edustaa mahdollisesti sisäistä tai käyttäjän määrittelemää tyyppiä. Ohjelmoija valitsee malliparametrille nimen. Päätimme esimerkissämme nimetä min():in malliparametrin Type:ksi, mutta olisimme voineet nimetä sen miksi tahansa:

```
template <class Glorp>
    Glorp min( Glorp a, Glorp b ) {
        return a < b ? a : b;
}
```

Kun mallista tehdään ilmentymä (*instantioidaan*), todellinen sisäinen tai käyttäjän määrittelemä tyyppi korvaa mallityyppiparametrin. Jokainen tyypeistä int, double, char*, vector<int> tai list<double>* on kelvollinen malliargumenttityypiksi.

Mallityypitön parametri muodostuu tavallisesta esittelystä. Se ilmaisee, että parametrin nimi edustaa mahdollista arvoa. Tämä arvo edustaa vakiota mallin määrittelyssä. Esimerkiksi size on mallityypitön parametri, joka on vakioarvo ja edustaa taulukon kokoa, johon arr viittaa:

```
template <class Type, int size>
    Type min( Type (&arr) [size] );
```

Kun funktiomalli min() instantioidaan, size:n arvo korvataan vakioarvolla, joka tiedetään käännöshetkellä.

Funktion määrittelyn tai esittelyn jälkeen tulee malliparametriluettelo. Paitsi se, että malliparametrit ovat kuin tyyppimääreitä tai vakioarvoja, mallifunktion määrittely näyttää samalta kuin tyypittömän funktion määrittely. Katsotaanpa esimerkkiä.

```
template <class Type, int size>
Type min( const Type (&r_array)[size] )
{
    /* parametroitu funktio, joka etsii
     * taulukon sisältämän minimiarvon */

    Type min_val = r_array[0];
    for ( int i = 1; i < size; ++i )
        if ( r_array[i] < min_val )
            min_val = r_array[i];

    return min_val;
}
```

Esimerkissämme Type ilmaisee min()-funktion paluutyyppiä, r_array-parametrinsa tyyppiä ja paikallisen min_val-muuttujan tyyppiä; size ilmaisee taulukon kokoa, johon r_array viittaa. Ohjelman aikana Type korvataan lukuisilla sisäisillä ja käyttäjän määrittelemillä tyypeillä ja size korvataan lukuisilla vakioarvoilla, joita min()-funktio käyttää varsinaisen tehtävänsä suorittamiseen. (Muista, että funktion käyttötapoja on kaksi: funktion käynnistäminen ja osoitteen ottaminen.) Tätä tyypin ja arvon korvaamista sanotaan *mallin instantioinniksi*. Katsomme mallin instantioimista seuraavassa kohdassa.

min()-funktiomallimme parametriluettelo saattaa näyttää hieman niukalta. Kuten kerroimme kohdassa 7.3, taulukkoparametri välitetään aina osoittimena taulukon ensimmäiseen elementtiin eikä taulukkoargumentin ensimmäistä ulottuvuutta tiedetä funktion määrittelyssä. Jotta helpottaisimme tätä ongelmaa, päätimme tässä esitellä min()-funktion parametrin viittauksena taulukkoon. Tämä ratkaisee ongelman niin, että käyttäjien ei tarvitse välittää toista argumenttia määrittääkseen taulukon koon, mutta haittapuoli on, että sen käyttö eri kokoisten int-taulukoiden kanssa generoi erilaisia ilmentymiä min()-funktiosta.

Malliparametrin nimeä voidaan käyttää sen jälkeen, kun se on esitelty ja siihen saakka, kun mallin esittely tai määrittely loppuu. Mallityyppiparametri toimii kuin tyyppimäärite mallin määrittelyn loppuun saakka; sitä voidaan käyttää täsmälleen samalla tavalla kuin sisäistä tai käyttäjän määrittelemää tyyppimäärettä, kuten muuttujaesittelyissä ja tyyppimuunnoksissa. Mallityypitön parametri toimii kuin vakioarvona mallin määrittelyn loppuun saakka; sitä voidaan käyttää, kun vakioarvoja tarvitaan, ehkäpä taulukkoesittelyn taulukon koon määrittämiseen tai luetellun vakion alkuarvona.

```
// size määrittää taulukkoparametrin koon ja
//     alustaa const int -arvon
template <class Type, int size>
    Type min( const Type (&r_array)[size] )
{
    const int loc_size = size;
    Type loc_array[loc_size];
    // ...
}
```

Jos oliolla, funktiolla tai tyypillä on sama nimi kuin malliparametrilla, joka on esitelty globaalilla viittausalueella, jää globaalilla viittausalueella oleva nimi piiloon. Seuraavassa esimerkissä tmp:n tyyppi ei ole double, vaan sen tyyppi on malliparametrin Type:

```
typedef double Type;
template <class Type>
    Type min( Type a, Type b )
{
    // tmp:n tyyppi on malliparametrin Type
    // eikä globaali typedef-tyyppi double
    Type tmp = a < b ? a : b;
    return tmp;
}
```

Oliolla tai tyypillä, joka on esitelty funktiomallin määrittelyssä, ei voi olla samaa nimeä kuin malliparametrilla:

```
template <class Type>
    Type min( Type a, Type b )
{
    // virhe: esittelee uudelleen malliparametrin Type
    typedef double Type;
    Type tmp = a < b ? a : b;
    return tmp;
}
```

Mallityyppiparametrin tyypin nimeä voidaan käyttää funktiomallin paluutyypin määrittämiseen:

```
// ok: T1 edustaa min()-funktion paluutyyppiä,
//     T2 ja T3 edustavat sen parametrityyppejä
template <class T1, class T2, class T3>
    T1 min( T2, T3 );
```

Malliparametrin nimeä voidaan käyttää vain kerran samassa malliparametriluettelossa. Esimerkiksi seuraava aiheuttaa käännöksenaikaisen virheen:

```
// virhe: malliparametrin nimeä Type ei saa käyttää uudelleen
template <class Type, class Type>
    Type min( Type, Type );
```

Malliparametrin nimeä voidaan käyttää kuitenkin uudelleen muissa funktiomallien esittelyissä ja määrittelyissä:

```
// ok: Type-nimen käyttö toisissa malleissa
template <class Type>
    Type min( Type, Type );

template <class Type>
    Type max( Type, Type );
```

Malliparametrien nimien ei tarvitse olla samoja mallin esittelyissä ja määrittelyissä. Esimerkiksi seuraavat kolme `min()`-funktion esittelyä viittaavat kaikki samaan funktiomalliin:

```
// kaikki kolme min()-funktion esittelyä
// viittaavat samaan funktiomalliin

// mallin jatkoesittelyitä
template <class T> T min( T, T );
template <class U> U min( U, U );

// mallin alkuperäinen määrittely
template <class Type>
    Type min( Type a, Type b ) { /* ... */ }
```

Ei ole olemassa rajoitusta sille, kuinka monta kertaa malliparametri voi esiintyä funktion parametriluettelossa. Seuraavassa esimerkissä käytetään `Type`:ä kahden erilaisen funktioparametrin tyypin esittämiseen:

```
#include <vector>
// ok: Type käytetty monta kertaa mallin parametriluettelossa
template <class Type>
    Type sum( const vector<Type> &, Type );
```

Jos funktiomallissa on useampi kuin yksi mallityyppiparametri, pitää jokaisen sellaisen edessä olla avainsana class tai typename.

```
// ok: avainsanat typename ja class ovat keskenään vaihdettavissa
template <typename T, class U>
    T minus( T*, U );

// virhe: pitää olla <typename T, class U> tai
//              <typename T, typename U>
template <typename T, U>
    T sum( T*, U );
```

Funktion malliparametriluettelossa on avainsanoilla typename ja class sama merkitys ja ne ovat keskenään vaihdettavissa. Niitä molempia voidaan käyttää eri mallityyppiparametrien esittelyyn samassa malliparametriluettelossa (kuten tehdään edellisessä funktiomallissa minus()). Tuntuu jotenkin selkeämmältä käyttää avainsanaa typename avainsanan class sijasta mallityyppiparametrin määrittämiseen, sillä avainsana typename selvästi ilmaisee, että sitä seuraava nimi on tyyppinimi. Kuitenkin avainsana typename lisättiin C++:aan vasta vähän aikaa sitten osaksi C++-standardia, jolloin vanhemmat ohjelmat käyttävät todennäköisesti yksinomaan avainsanaa class. (Mainitsemattakin on selvää, että avainsana class on lyhempi kuin typename ja koska ihmiset tekevät niin kuin tekevät, niin...)

Avainsana typename lisättiin C++:aan, jotta mallien määrittelyjä voitaisiin jäsentää. Tämä aihe on mennyt hieman eteenpäin ja kerromme tässä vain lyhyesti, miksi avainsanaa typename tarvitaan. Niille, jotka haluavat tietää enemmän, suosittelemme Stroustrupin kirjan *Design and Evolution of C++* tutkimista.

Jotta mallimäärittelyä voitaisiin jäsentää, pitää kääntäjän erottaa toisistaan lausekkeet, jotka ovat tyyppejä ja ne, jotka eivät ole. Kääntäjälle ei ole aina mahdollista yksilöidä, mitkä lausekkeet ovat tyyppejä mallin määrittelyssä. Jos esimerkiksi kääntäjä havaitsee lausekkeen Parm::name mallin määrittelyssä ja jos Parm on mallityyppiparametri, joka edustaa luokkaa, viittaako name silloin Parm:in tyyppijäseneen?

```
template <class Parm, class U>
    Parm minus( Parm* array, U value )
{
    Parm::name * p; // Onko tämä osoittimen esittely vai
            // kertolasku? Kertolasku.
}
```

Kääntäjä ei tiedä, onko name tyyppi, koska se ei voi katsoa luokan määrittelystä, jota Parm edustaa, ennen kuin malli on instantioitu. Jotta mallimäärittely voitaisiin jäsentää omin neuvoin, pitää käyttäjien ilmaista kääntäjälle, mitkä lausekkeet ovat tyyppilausekkeita. Meka-

nismi, jolla kääntäjälle kerrotaan, että lauseke on tyyppilauseke, on laittaa avainsana typename lausekkeen eteen. Jos esimerkiksi haluaisimme lausekkeen Parm::name olevan tyyppinimi funktiomallissa minus() ja siten saada koko lausekkeesta osoitinesittelyn, muokkaisimme sitä seuraavasti:

```
template <class Parm, class U>
    Parm minus( Parm* array, U value )
{
    typename Parm::name * p; // ok: osoitinesittely
}
```

Avainsanaa typename voidaan käyttää myös malliparametriluettelossa ilmaisemaan, milloin malliparametri on tyyppi.

Funktiomalli voidaan esitellä inline (välittömänä) tai extern (ulkoisena) samaan tapaan kuin malliton funktio. Määre sijoitetaan malliparametriluettelon jälkeen eikä avainsanan template eteen.

```
// ok: avainsana tulee malliparametriluettelon jälkeen
template <typename Type>
    inline
    Type min( Type, Type );

// virhe: virheellinen inline-määreen sijoitus
inline
template <typename Type>
    Type min( Array<Type>, int );
```

Harjoitus 10.1

Yksilöi, mitkä seuraavista funktiomallien määrittelyistä ovat virheellisiä. Korjaa jokainen, jonka toteat virheelliseksi.

```
(a) template <class T, U, class V>
     void foo( T, U, V );

(b) template <class T>
     T foo( int *T );

(c) template <class T1, typename T2, class T3>
     T1 foo( T2, T3 );

(d) inline template <typename T>
     T foo( T, unsigned int* );

(e) template <class myT, class myT>
     void foo( myT, myT );

(f) template <class T>
     foo( T, T );
```

(g)
```
typedef char Ctype;
template <class Ctype>
  Ctype foo( Ctype a, Ctype b );
```

Harjoitus 10.2

Mitkä seuraavista mallien uudelleenesittelyistä, jos yksikään, ovat virheellisiä? Miksi?

(a)
```
template <class Type>
  Type bar( Type, Type );

template <class Type>
  Type bar( Type, Type );
```

(b)
```
template <class T1, class T2>
  void bar( T1, T2 );

template <typename C1, typename C2>
  void bar( C1, C2 );
```

Harjoitus 10.3

Kirjoita uudelleen kohdassa 7.3.3 esitelty putValues()-funktio funktiomalliksi. Parametroi funktiomalli niin, että siinä on kaksi malliparametria (yksi taulukkoelementin tyypille ja yksi taulukon koolle) ja yksi funktioparametri, joka on viittaus taulukkoon. Tee myös funktiomallin määrittely.

10.2 Funktiomallin instantiointi

Funktiomalli määrittää, kuinka yksittäiset funktiot voidaan rakentaa annetuilla yhdellä tai useammalla todellisella tyypillä tai arvolla. Tätä rakennusprosessia kutsutaan *mallin instantioinniksi*. Se tapahtuu väistämättä sivuvaikutuksena funktiomallin käynnistämisessä tai sen osoitteen ottamisessa. Esimerkiksi seuraavassa ohjelmassa min() instantioidaan kahdesti: kerran viisielementtisen int-taulukon tyypin vuoksi ja kerran kuusielementtisen double-taulukon tyypin vuoksi.

```
// funktiomallin min() määrittely
// tyyppiparametrilla Type ja tyypittömällä parametrilla size

template <typename Type, int size>
    Type min( Type (&r_array)[size] )
{
    Type min_val = r_array[0];
    for ( int i = 1; i < size; ++i )
        if ( r_array[i] < min_val )
            min_val = r_array[i];
```

```
    return min_val;
}

// size määrittämättä -- ok
// size = alustusluettelossa olevien arvojen lukumäärä
int ia[] = { 10, 7, 14, 3, 25 };

double da[6] = { 10.2, 7.1, 14.5, 3.2, 25.0, 16.8 };

#include <iostream>
int main()
{
    // min()-funktion instantiointi taulukolla, jossa on 5 int-tyyppiä,
    // jossa Type => int, size => 5
    int i = min( ia );
    if ( i != 3 )
        cout << "??oops: integer min() failed\n";
    else cout << "!!ok: integer min() worked\n";


    // min()-funktion instantiointi taulukolla, jossa on 6 double-tyyppiä
    // with Type => double, size => 6
    double d = min( da );
    if ( d != 3.2 )
        cout << "??oops: double min() failed\n";
    else cout << "!!ok: double min() worked\n";
    return 0;
}
```

Kutsu

```
int i = min( ia );
```

saa aikaan seuraavan min()-funktion kokonaislukuesiintymän instantioinnin, jossa Type korvataan tyypillä int ja size arvolla 5:

```
int min( int (&r_array)[5] )
{
    int min_val = r_array[0];
    for ( int ix = 1; ix < 5; ++ix )
        if ( r_array[ix] < min_val )
            min_val = r_array[ix];
    return min_val;
}
```

Samalla tavalla kutsu

```
double d = min( da );
```

saa aikaan min()-ilmentymän instantioinnin, jossa Type korvataan tyypillä double ja size arvolla 6.

Sekä tyyppiparametria Type että tyypitöntä parametria size on käytetty funktioparametrin tyyppinä. Jotta voitaisiin päätellä malliargumenttina käytettävä todellinen tyyppi ja arvo, funktiokutsussa annetun argumentin tyyppi tutkitaan. Esimerkissämme tyyppejä ia (tarkoittaa taulukkoa, jossa on viisi int-tyyppiä) ja da (tarkoittaa taulukkoa, jossa on kuusi double-tyyppiä) käytetään malliargumenttien päättelyyn joka instantioinnissa. Prosessia, jossa päätellään malliargumenttien tyypit ja arvot funktioargumenteista, kutsutaan *malliargumenttien päättelyksi*. Katsomme yksityiskohtaisemmin malliargumenttien päättelyä seuraavassa kohdassa. (Ei ole myöskään mahdollista luottaa malliargumenttien päättelyyn, vaan pitää määrittää malliargumentit eksplisiittisesti. Katsomme kohdassa 10.4, kuinka se tehdään.)

Funktiomalli instantioidaan joko silloin, kun se käynnistetään, tai silloin, kun sen osoite otetaan. Seuraavassa esimerkissä alustetaan pf-osoitin funktiomallin instantioinnin osoitteella. Instantioinnin malliargumentit päätellään tutkimalla funktion parametrityyppi, johon pf viittaa.

```
template <typename Type, int size>
    Type min( Type (&p_array)[size] ) { /* ... */ }

// pf osoittaa:  int min( int (&)[10] )
int (*pf)(int (&)[10]) = &min;
```

pf-osoittimen tyyppi on osoitin funktioon parametrityypillä int(&)[10]. Tämän parametrin tyyppi määrää malliargumentin tyypin Type:lle ja malliargumentin arvon size:lle, kun min() instantioidaan. Type:n malliargumentti on int ja size:n malliargumentti on 10. Instantioitu funktio on min(int(&)[10]) ja pf-osoitin viittaa mallin tähän instantiointiin.

Kun funktiomallin instantioinnin osoite otetaan, tilanteen pitää olla sellainen, että malliargumentti voidaan päätellä yksilöllisestä tyypistä tai arvosta. Ellei yksilöllistä tyyppiä tai arvoa voida päätellä, se saa aikaan käännöksenaikaisen virheen. Esimerkiksi:

```
template <typename Type, int size>
    Type min( Type (&r_array)[size] ) { /* ... */ }

typedef int (&rai)[10];
typedef double (&rad)[20];

void func( int (*)(rai) );
void func( double (*)(rad) );

int main() {
```

```
    // virhe: mikä min():in instantiointi?
    func( &min );
}
```

Funktio func() on ylikuormitettu eikä ole mahdollista, että voitaisiin päätellä malliargumentille Type yksilöllinen tyyppi katsomalla funktion parametrin tyyppiä, eikä yksilöllistä arvoa malliargumentille size. Funktion func() kutsuminen voisi instantioida kumman tahansa seuraavista funktioista:

```
min( int (*)(int(&)[10]) )
min( double (*)(double(&)[20]) )
```

Koska ei ole mahdollista ilmaista yksilöllistä instantiointia argumentille func()-funktioon, osoitteen ottaminen funktiomallin instantioinnista tässä yhteydessä saa aikaan käännöksenaikaisen virheen.

Tämä käännöksenaikainen virhe voidaan eliminoida, jos käytämme tyyppimuunnosta eksplisiittisesti ilmaistaksemme argumentin tyypin:

```
int main() {
    // ok: tyyppimuunnos ilmaisee argumentin tyypin
    func( static_cast< double(*)(rad) >(&min) );
}
```

Parempi ratkaisu on käyttää eksplisiittisiä malliargumentteja kuten kuvasimme kohdassa 10.4.

10.3 Malliargumentin päättely

Kun funktiomallia kutsutaan, malliargumenttien tyypit ja arvot päätellään tutkimalla funktioargumenttien tyyppejä. Tätä prosessia kutsutaan *malliargumenttien päättelyksi*.

Funktiomallin min() funktioparametri on viittaus taulukkoon, jonka tyyppi on Type:

```
template <class Type, int size>
    Type min( Type (&r_array)[size] ) { /* ... */ }
```

Jotta funktion argumentti vastaisi funktion parametria, pitää argumentin olla myös lvalue, joka edustaa taulukkotyyppiä. Seuraava kutsu on virheellinen, koska pval on tyyppiä int* eikä int-kokonaislukujen taulukkotyyppinen lvalue.

```
void f( int pval[9] ) {
    // virhe: Type (&)[] != int*
    int jval = min( pval );
}
```

Funktiomallin instantioinnin paluutyyppiä ei huomioida, kun malliargumenttien tyypit päätellään malliargumenttien päättelyn aikana. Esimerkiksi min()-funktion kutsussa, joka kirjoitetaan

```
double da[8] = { 10.3, 7.2, 14.0, 3.8, 25.7, 6.4, 5.5, 16.8 };
int i1 = min( da );
```

on min():in instantioinnin parametri tyyppiä osoitin kahdeksan double-tyypin taulukkoon. Tämän instantioinnin palauttama arvo on double-tyyppi. Paluuarvo konvertoidaan int-tyypiksi, ennen kuin i1 alustetaan. Vaikka min()-kutsun tulosta käytetään int-tyyppisen olion alustamiseen, se ei vaikuta malliargumentin päättelyyn.

Jotta funktioargumentin päättely onnistuisi, ei funktioargumentin tyypin tarvitse olla täsmälleen vastaavan funktioparametrin tyyppinen. Seuraavat kolme erilaista tyyppikonversiota sallitaan: lvalue-muunnos, määrekonversio ja konversio kantaluokkaan, joka on instantioitu luokkamallista. Katsokaamme niitä jokaista vuorollaan.

Muista, että lvalue-muunnos on joko lvalue rvalueksi -konversio, taulukko osoittimeksi -konversio tai funktio osoittimeksi -konversio. (Nämä konversiot esiteltiin kohdassa 9.3.) Jotta voisimme kuvata, kuinka lvalue-muunnos vaikuttaa malliargumentin päättelyyn, miettikäämme min2()-funktiota, jolla on yksi malliparametri nimeltään Type ja kaksi funktioparametria. min2()-funktion ensimmäinen parametri on osoitin tyyppiä Type*. size ei ole enää malliparametri kuten oli min()-määrityksessä, vaan siitä on tullut funktioparametri, jolle pitää määrittää arvo eksplisiittisesti funktioargumentilla, kun min2():ta kutsutaan:

```
template <class Type>
    // ensimmäinen parametri on Type*
    Type min2( Type* array, int size )
{
    Type min_val = array[0];
    for ( int i = 1; i < size; ++i )
        if ( array[i] < min_val )
            min_val = array[i];

    return min_val;
}
```

min2():ta voidaan kutsua ensimmäisellä argumentilla, joka on tyyppiä taulukko, jossa on neljä int-kokonaislukua kuten seuraavassa esimerkissä:

```
int ai[4] = { 12, 8, 73, 45 };

int main() {
    int size = sizeof (ai) / sizeof (ai[0]);

    // ok: taulukko osoittimeksi -konversio
    min2( ai, size );
}
```

Funktioargumentin ai tyyppi on neljän int-tyypin taulukko eikä täysin vastaa funktion parametrin Type*-tyyppiä. Koska kuitenkin taulukko osoittimeksi -konversio sallitaan, konvertoidaan ai-argumentti int*-tyypiksi ennen kuin malliargumentti Type päätellään. Type:n malliargumentiksi päätellään sitten int ja instantioitu funktiomalli on min2(int*,int).

Määrekonversio lisää const- tai volatile-määreet osoittimiin (määrekonversiot ovat esiteltyinä myös kohdassa 9.3). Jotta voisimme kuvata, kuinka määrekonversiot vaikuttavat malliargumentin päättelyyn, miettikäämme min3()-funktiota, jonka ensimmäinen parametri on tyyppiä const Type*:

```
template <class Type>
    // ensimmäinen parametri on const Type*
    Type min3( const Type* array, int size ) {
    // ...
}
```

min3()-funktiota voidaan kutsua ensimmäisellä argumentilla, jonka tyyppi on int* kuten seuraavassa esimerkissä:

```
int *pi = &ai;
// ok: määrekonversio tyypiksi const int*
int i = min3( pi, 4 );
```

Funktioargumentin pi tyyppi on osoitin int-tyyppiin, eikä se täysin vastaa funktioparametrin tyyppiä const Type*. Koska määrekonversio on sallittu, funktioargumentti konvertoidaan tyypiksi const int* ennen kuin malliargumentti päätellään. Type:n malliargumentiksi on siten päätelty int ja instantioitu funktiomalli on min3(const int*,int).

Katsokaamme nyt konversiota kantaluokkaan, joka on instantioitu luokkamallista. Malliargumentin päättely voi edetä, jos funktioparametrin tyyppi on luokkamalli ja jos argumentti on luokka, joka on instantioitu funktioparametrissa määritetystä luokkamallista. Jotta voisimme kuvata tätä konversiota, miettikäämme uutta funktiomallia nimeltään min4(), jolla on parametri tyyppiä Array<Type>& (jossa Array on luokkamalli, joka määriteltiin kohdassa 2.5). (Luvussa 16 esitetään kaikki luokkamalleista.)

```
template <class Type>
    class Array { /* ... */ };

template <class Type>
    Type min4( Array<Type>& array )
{
    Type min_val = array[0];
    for ( int i = 1; i < array.size(); ++i )
        if ( array[i] < min_val )
            min_val = array[i];

    return min_val;
}
```

min4()-funktiota voidaan kutsua ensimmäisellä argumentilla, joka on tyyppiä ArrayRC<int> kuten seuraavassa esimerkissä. (ArrayRC on luokkamalli, joka on myös määritelty luvussa 2; luokan periytymistä käsitellään yksityiskohtaisesti luvuissa 17 ja 18.)

```
template <class Type>
    class ArrayRC : public Array<Type> { /* ... */ };

int main() {
    ArrayRC<int> ia_rc(10);
    min4( ia_rc );
}
```

Funktion argumentti ia_rc on tyyppiä ArrayRC<int>. Se ei täysin vastaa funktioparametrin tyyppiä Array<Type>&. Koska eräs ArrayRC<int>-luokan kantaluokista on Array<int> ja koska Array<int> on instantioitu funktioparametrissa määritetystä luokkamallista ja koska luokkatyypistä johdettua funktion argumenttia voidaan käyttää malliargumentin päättelyyn, konvertoidaan funktion argumentti ArrayRC<int> Array<int>-tyypiksi ennen kuin malliargumentti päätellään. Type:n malliargumentin päätellään siten olevan int, ja instantioitu funktiomalli on min4(Array<int>&).

Malliargumentin päättelyyn voidaan ottaa mukaan useampi kuin yksi funktioargumentti. Jos malliparametri esiintyy useita kertoja funktion parametriluettelossa, pitää jokaisen päätellyn tyypin vastata malliargumentista pääteltyä ensimmäistä tyyppiä. Esimerkiksi:

```
template <class T> T min5( T, T ) { /* ... */ }
unsigned int ui;

int main() {
    // virhe: ei voi instantioida tätä: min5( unsigned int, int )
    // pitää olla: min( unsigned int, unsigned int ) tai
    //         min( int, int )
    min5( ui, 1024 );
}
```

Kun kutsutaan min5()-funktiota, pitää molempien argumenttien olla samantyyppisiä — kumpikin joko int tai molemmat unsigned int — koska malliparametri T pitää sitoa yhteen tyyppiin. T:n malliargumentti, joka on päätelty funktion ensimmäisestä argumentista, on int. T:n malliargumentti, joka on päätelty funktion toisesta argumentista, on unsigned int. Koska T:n malliargumentin tyyppi on päätelty erilaiseksi kummallekin funktioargumentille, malliargumentin päättely epäonnistuu ja mallin instantiointi on virhe. (Eräs tapa tämän kiertämiseen on määrittää eksplisiittisesti malliargumentit, kun min5()-funktio käynnistetään. Katsomme kohdassa 10.4, kuinka tämä tehdään.)

Mahdollisten tyyppikonversioiden rajoitus koskee ainoastaan funktion argumentteja, jotka osallistuvat malliargumentin päättelyyn. Kaikille muille argumenteille sallitaan kaikki tyyppikonversiot. Seuraavalla sum()-funktiomallilla on kaksi parametria. Ensimmäisen parametrin op1-argumentti ottaa osaa Type-malliargumentin päättelyyn. Toisen parametrin op2-argumentti

ei ota osaa.

```
template <class Type>
    Type sum( Type op1, int op2 ) { /* ... */ }
```

Koska toinen argumentti ei ota osaa malliargumentin päättelyyn, mitä tahansa tyyppikonversiota voidaan käyttää toiseen argumenttiin, kun sum()-funktiomallin ilmentymää kutsutaan. (Tyyppikonversiot, joita funktion argumenttiin voidaan käyttää, on kuvattu kohdassa 9.3.) Esimerkiksi:

```
int ai[] = { ... };
double dd;
int main() {
    // sum( int, int ) instantioidaan
    sum( ai[0], dd );
}
```

Funktion toisen argumentin, dd:n, tyyppi ei vastaa funktion int-parametria. Tämä kutsu sum()-funktiomallin ilmentymään ei kuitenkaan ole virhe, koska toisen argumentin tyyppi on kiinteä eikä riipu malliparametrista. Funktio sum(int,int) instantioidaan kutsua varten. Argumentti dd konvertoidaan int-tyypiksi käyttämällä liukuluku-kokonaisvakiokonversiota.

Malliargumentin päättelyn yleinen algoritmi on siten seuraava:

1. Jokainen funktion argumentti tutkitaan vuorollaan ja katsotaan, löytyykö vastaavaa malliparametria funktioparametrityypeistä.
2. Jos malliparametri löytyy, vastaava malliargumentti päätellään tutkimalla funktioargumentin tyyppi.
3. Funktioparametrin ja funktioargumentin tyyppien ei tarvitse vastata toisiaan täsmällisesti. Seuraavia tyyppikonversioita voidaan käyttää funktioargumenttiin, jotta siitä saataisiin vastaava funktioparametrin tyyppi:
   - Lvalue-muunnokset
   - Määrekonversiot
   - Johdettu luokka kantaluokan tyypiksi -konversio edellyttäen, että funktioparametrin muoto on T<args>, T<args>& tai T<args>*, jossa parametriluettelo args sisältää vähintään yhden malliparametreista.
4. Jos sama malliparametri löytyy useammasta kuin yhdestä funktioparametrista, pitää jokaisen vastaavasta funktioargumentista päätellyn malliargumentin olla samanlainen.

Harjoitus 10.4

Nimeä kaksi tyyppikonversiota, jotka on sallittu funktioargumenteille malliargumentin päättelyssä.

Harjoitus 10.5

Olkoon määriteltynä seuraavat mallit:

```
template <class Type>
    Type min3( const Type* array, int size ) { /* ... */ }
template <class Type>
    Type min5( Type p1,Type p2 ) { /* ... */ }
```

Mitkä seuraavista kutsuista ovat virheellisiä, vai onko yksikään? Miksi?

```
double dobj1, dobj2;
float fobj1, fobj2;
char cobj1, cobj2;
int ai[5] = { 511, 16, 8, 63, 34 };

(a) min5( cobj2, 'c' );
(b) min5( dobj1, fobj1 );
(c) min3( ai, cobj1 );
```

10.4 Eksplisiittiset malliargumentit ◆A

Joissakin tilanteissa ei ole mahdollista päätellä malliargumenttien tyyppiä. Kuten näimme edellisessä kohdassa min5()-funktiomallin yhteydessä, että jos malliargumentin päättelyn tuloksena on kaksi eri tyyppiä samalle malliparametrille, kääntäjä antaa virheilmoituksen, että malliargumentin päättely on epäonnistunut.

Sellaisissa tilanteissa on välttämätöntä ohittaa malliargumentin päättelyn mekanismi ja *määrittää eksplisiittisesti* käytettävät malliargumentit. Malliargumentit määritetään eksplisiittisesti pilkuin eroteltuun luetteloon ja kulmasulkujen sisälle (< >). Malliargumentin jälkeen tulee funktiomallin instantioinnin nimi. Jos oletetaan esimerkiksi, että aikaisemmin käyttämämme min5()-funktion yhteydessä T:n malliargumentin tulisi olla unsigned int, kutsu min5()-funktiomallin instantiointiin voitaisiin kirjoittaa seuraavasti:

```
// min5( unsigned int, unsigned int ) instantioitu
min5< unsigned int >( ui, 1024 );
```

Tässä tapauksessa malliargumenttiluettelo, <unsigned int>, määrittää eksplisiittisesti malliargumentin tyypin. Koska malliargumentti on tunnettu, ei funktion kutsu enää ole virheellinen.

Huomaa, että min5()-funktion kutsun toinen funktioargumentti on 1024, jonka tyyppi on int. Koska toinen funktioparametri on kiinnitetty tyypiksi unsigned int eksplisiittisellä malliargumentilla, konvertoidaan toinen funktioargumentti unsigned int -tyypiksi käyttämällä kokonaisvakiokonversiota.

Näimme edellisessä kohdassa, että malliargumentin päättelyn yhteydessä on sallittu vain rajoitettu joukko tyyppikonversioita. Kokonaisvakiokonversio int-tyypistä unsigned int -tyypiksi ei ole yksikään sallituista konversioista. Mutta silloin, kun malliargumentit määritetään eksplisiittisesti, ei ole tarvetta päätellä malliargumentteja. Funktioparametrien tyypit ovat siten

kiinnitettyjä. Kun malliargumentit on määritetty eksplisiittisesti, mitä tahansa implisiittistä tyyppikonversiota voidaan käyttää, kun halutaan konvertoida funktioargumentti vastaavan funktioparametrin tyyppiseksi.

Sen lisäksi, että funktioargumenteille sallitaan tyyppikonversioita, tarjoavat eksplisiittiset malliargumentit ratkaisuja muihinkin ohjelmointiongelmiin. Mietitäänpä seuraavaa ongelmaa. Haluamme määritellä funktiomallin nimeltään sum() niin, että funktio, joka instantioidaan tästä mallista, palauttaisi tyypin, joka olisi tarpeeksi suuri sisältämään kahden minkä tahansa tyypin arvojen summan. Tyypit voitaisiin välittää missä tahansa järjestyksessä. Kuinka tekisimme sen? Kuinka määrittäisimme funktiomallin sum() paluutyypin?

```
// T tai U paluutyyppinä?
template <class T, class U>
    ??? sum( T, U );
```

Tapauksessamme vastaus on, että emme käytä kumpaakaan parametria. Kummankin parametrin käyttäminen on tuomittu epäonnistumaan jossain vaiheessa:

```
char ch; unsigned int ui;

// kumpikaan, T eikä U, toimi paluutyyppinä
sum( ch, ui ); // ok: U sum( T, U );
sum( ui, ch ); // ok: T sum( T, U );
```

Ratkaisu on esitellä kolmas malliparametri, joka yksinkertaisesti määrittelee funktion paluutyypin.

```
// T1 ei esiinny funktion parametriluettelossa
template <class T1, class T2, class T3>
    T1 sum( T2, T3 );
```

Koska paluutyyppi voi olla muun tyyppinen kuin funktioargumenttien tyypit, ei T1:tä mainita funktioparametriluettelossa. Tämä saattaa olla ongelma, koska T1:n malliargumenttia ei voida päätellä funktioargumenteista. Jos kuitenkin laitamme eksplisiittiset malliargumentit sum()-funktion kutsun instantiointiin, vältämme kääntäjän virheilmoituksen, että T1:n malliargumenttia ei voida päätellä. Esimerkiksi:

```
typedef unsigned int ui_type;
ui_type calc( char ch, ui_type ui ) {

    // ...
    // virhe: T1:tä ei voida päätellä
    ui_type loc1 = sum( ch, ui );

    // ok: malliargumentit eksplisiittisesti määritetty
    // T1 ja T3 ovat tyyppiä unsigned int, T2 on char
    ui_type loc2  = sum< ui_type, char, ui_type >( ch, ui );
}
```

Se, mitä todella haluaisimme tehdä, on määrittää eksplisiittinen malliargumentti T1:lle ja jättää pois eksplisiittiset argumentit tyypeiltä T2 ja T3, koska nämä malliargumentit voidaan päätellä kutsun funktioargumenteista.

Kun eksplisiittistä määritystä käytetään, pitää luetella vain ne malliargumentit, joita ei voida implisiittisesti päätellä sillä rajoituksella, että kuten oletusargumenteilla, voimme jättää pois vain niiden jälkeen tulevat argumentit. Esimerkiksi:

```
// ok: T3 on unsigned int
// T3 päätellään ui:n  tyypistä
ui_type loc3 = sum< ui_type, char >( ch, ui );

// ok: T2 on char ja T3 on unsigned int
// T2 ja T3 on päätelty pf:n tyypistä
ui_type (*pf)( char, ui_type ) = &sum< ui_type >;

// virhe: vain jäljessä olevat argumentit voidaan jättää pois
ui_type loc4 = sum< ui_type, , ui_type >( ch, ui );
```

Muissa tilanteissa ei ole mahdollista päätellä malliargumentteja asiayhteydestä, jossa funktiomallin instantiointia on käytetty. Sellaisissa asiayhteyksissä on mahdotonta käyttää funktiomallin instantiointia ilman eksplisiittisiä malliargumentteja. Näiden tilanteiden havaitseminen ja tarve tukea niitä johtivat siihen, että eksplisiittisiä malliargumentteja alettiin tukea C++-standardissa. Seuraavassa esimerkissä otetaan sum()-funktiomallin instantioinnin osoite ja välitetään argumenttina kutsussa ylikuormitetulle manipulate()-funktiolle. Kuten näimme kohdassa 10.2, ei ole mahdollista valita sum()-funktion instantiointia ja välittää sitä argumenttina vain katsomalla manipulate()-funktioiden parametriluetteloita. Kutsun tyydyttämiseksi voitaisiin instantioida kaksi eri sum()-mallifunktiota. Funktion manipulate() kutsu on *moniselitteinen*. Eräs ratkaisu poistaa moniselitteisyys on tehdä eksplisiittinen tyyppimuunnos. Parempi ratkaisu on eksplisiittisten malliargumenttien käyttö. Eksplisiittiset malliargumentit ilmaisevat, mitä instantiointia sum()-funktiosta on käytetty ja mitä manipulate()-funktiota on kutsuttu. Esimerkiksi:

```
template <class T1, class T2, class T3>
    T1 sum( T2 op1, T3 op2 ) { /* ... */ }

void manipulate( int (*pf)( int,char ) );
void manipulate( double (*pf)( float,float ) );
```

```
int main( )
{
    // virhe: mikä sum-instantiointi?
    // int sum( int, char )  vai
    // double sum( float, float )  ?
    manipulate( &sum );

    // ottaa instantioinnin osoitteen:
    // double sum( float, float )
    // kutsuu: void manipulate( double (*pf)( float, float ) );
    manipulate( &sum< double, float, float > );
}
```

Meidän pitää mainita, että eksplisiittisiä malliargumentteja tulisi käyttää vain silloin, kun on ehdottoman välttämätöntä ratkaista moniselitteisyyksiä tai käyttää funktiomallien instantiointeja yhteyksissä, joissa malliargumentteja ei voida päätellä. Ensiksikin, on aina helpompaa jättää kääntäjän pääteltäväksi malliargumenttien tyypit ja arvot. Toiseksi, jos muokkaamme ohjelmiemme esittelyitä niin, että funktioargumenttien tyypit muuttuvat funktiomallin instantiointia kutsuttaessa, kääntäjä instantioi automaattisesti funktiomallin erilaisilla malliargumenteilla tekemällä jotain tai jättämällä tekemättä mitään. Jos toisaalta määritämme eksplisiittiset malliargumentit, pitää varmistua, että eksplisiittiset malliargumentit ovat yhä sopivia funktioargumenttien uusille tyypeille. Siitä syystä suosittelemme, että jätät eksplisiittiset malliargumentit pois aina, kun se on mahdollista.

Harjoitus 10.6

Nimeä kaksi tilannetta, joissa eksplisiittisten malliargumenttien käyttö on välttämätöntä.

Harjoitus 10.7

Olkoon määritelty seuraava funktiomalli sum():

```
template <class T1, class T2, class T3>
    T1 sum( T2, T3 );
```

Mitkä seuraavista kutsuista ovat virheellisiä, vai onko yksikään? Miksi?

```
double dobj1, dobj2;
float fobj1, fobj2;
char cobj1, cobj2;

(a) sum( dobj1, dobj2 );
(b) sum<double,double,double>( fobj1, fobj2 );
(c) sum<int>( cobj1, cobj2 );
(d) sum<double, ,double>( fobj2, dobj2 );
```

10.5 Mallin käännösmallit

Funktiomallin määrittely toimii kuin ohjeena lukemattomien funktioinstantiointien määrittelyille. Itse malli ei aiheuta yhdenkään funktion määrittelyä. Kun toteutus näkee esimerkiksi mallimäärittelyn

```
template <typename Type>
    Type min( Type t1, Type t2 )
{
    return t1 < t2 ? t1 : t2;
}
```

se tallettaa min()-funktiomallin sisäiseen esitystapaan, mutta se ei saa aikaan mitään muuta. Myöhemmin, kun toteutus näkee todellisen min()-funktiomallin käyttötilanteen kuten

```
int i, j;
double dobj = min( i, j );
```

se instantioi sitten min()-kokonaislukumäärittelyn mallimäärittelystä.

Tämä tuo esille useita kysymyksiä. Jotta kääntäjä kykenisi instantioimaan funktiomallin, pitääkö mallin määrittelyn olla näkyvissä, kun instantiointia kutsutaan? Pitääkö esimerkiksi min()-funktiomallin määrittely esiintyä ennen kuin sen kokonaislukuilmentymää käytetään dobj:n määrittelyyn? Sijoitammeko funktiomallien määrittelyt otsikkotiedostoihin (kuten teemme välittömien funktioiden määrittelyille), jotta ne voitaisiin ottaa mukaan kaikkialle sinne, missä funktiomallien instantiointeja käytetään? Vai voimmeko laittaa funktiomallien esittelyt vain otsikkotiedostoihin ja mallien määrittelyt tekstitiedostoihin (kuten teemme muille kuin välittömille funktioille)?

Jotta näihin kysymyksiin voitaisiin vastata, pitää kertoa C++:n *mallin käännösmallista* (*template compilation model*), joka määrittelee vaatimukset tavasta, jolla malleja määrittelevät ja käyttävät ohjelmat pitää järjestää. C++ tukee kahta mallin käännösmallia: *mukaan ottava käännösmalli* ja *erotteleva käännösmalli*. Tämän kohdan loppuosa kuvaa molempia käännösmalleja ja kertoo, kuinka niitä käytetään.

10.5.1 Mukaan ottava käännösmalli

Mukaan ottavassa käännösmallissa laitamme funktiomallin määrittelyn jokaiseen tiedostoon, jossa se instantioidaan. Teemme sen usein sijoittamalla funktiomallin määrittelyn otsikkotiedostoon, kuten teemme välittömille funktioille. Tämän muodon olemme valinneet käytettäväksi tässä kirjassa. Esimerkiksi:

```
// model1.h:
// mukaan ottava käännösmalli:
// mallin määrittelyt on sijoitettu otsikkotiedostoon

template <typename Type>
   Type min( Type t1, Type t2 ) {
```

```
    return t1 < t2 ? t1 : t2;
}
```

Tämä otsikkotiedosto otetaan mukaan jokaiseen tiedostoon, jossa min():in instantiointia on käytetty. Esimerkiksi:

```
// mallin määrittelyt on otettu mukaan ennen kuin
// mallin instantiointia on käytetty
#include "model1.h"

int i, j;
double dobj = min( i, j );
```

Otsikkotiedosto voidaan ottaa mukaan useaan ohjelmamme tiedostoon. Tarkoittaako tämä sitä, että kääntäjän pitää instantioida min():in kokonaislukuilmentymä jokaisessa tiedostossa, jossa kutsutaan instantiointia? Ei. Ohjelman pitää käyttäytyä niin kuin min():in kokonaislukuilmentymä olisi instantioitu vain kerran. Kuitenkin se, milloin ja missä instantiointi varsinaisesti tapahtuu, riippuu toteutuksesta. Tässä vaiheessa, mitä meihin tulee, pitää tietää vain, että min():in kokonaislukuilmentymä instantioidaan jossain kohtaa ohjelmaamme. (Kuten tulemme näkemään tämän kohdan loppuun mennessä, on mahdollista määrittää, milloin ja missä instantiointi tapahtuu käyttämällä eksplisiittistä instantiointiesittelyä. Sellaisia esittelyitä pitää joskus käyttää tuotantokehityksen myöhemmissä vaiheissa sovelluksen suorituskyvyn parantamiseksi.)

Funktiomallien sijoittamisessa otsikkotiedostoon on olemassa joitakin haittapuolia. Funktiomallin rungossa on kuvauksia toteutuksen yksityiskohdista, joita käyttäjämme ehkä haluaisivat jättää huomiotta tai haluaisimme piilottaa käyttäjiltä. Itse asiassa, jos funktiomalliemme määrittelyt ovat suuria, otsikkotiedostossa näkyvät toteutuksen pienet yksityiskohdat saattavat olla hämmentäviä. Lisäksi saman funktiomallin määrittelyiden kääntäminen useissa tiedostoissa voi lisätä tarpeettomasti ohjelmiemme käännösaikaa. Erotteleva käännösmalli mahdollistaa sen, että voimme erotella funktiomalliemme esittelyt ja määrittelyt. Katsokaamme, kuinka voisimme käyttää sitä.

10.5.2 Erotteleva käännösmalli

Erottelevassa käännösmallissa funktiomallien esittelyt sijoitetaan otsikkotiedostoon ja niiden määrittelyt ohjelmatekstitiedostoon. Tässä mallissa funktiomallien esittelyt ja määrittelyt on järjestetty samaan tapaan, kuten menetellään muiden kuin välittömien funktioiden esittelyiden ja määrittelyiden kanssa ohjelmissamme. Esimerkiksi:

```
// model2.h
// erotteleva käännösmalli:
// vain mallin esittely mukana
```

```
template <typename Type> Type min( Type t1, Type t2 );

// model2.C
// mallin määrittely
export template <typename Type>
    Type min( Type t1, Type t2 ) { /* ...*/ }
```

Ohjelman, joka käyttää min()-mallifunktion instantiointia, tarvitsee ottaa mukaan vain otsikkotiedosto ennen instantioinnin käyttöä:

```
// user.C
#include "model2.h"

int i, j;
double d = min( i, j ); // ok: käyttötilanne, joka vaatii instantioinnin
```

Vaikka min()-mallimäärittely ei ole näkyvissä user.C-tiedostossa, mallin instantiointia min(int,int) voidaan siitä huolimatta kutsua tässä tiedostossa. Jotta tämä kuitenkin olisi mahdollista, pitää min()-malli määritellä erityisellä tavalla. Huomaatko, kuinka? Jos katsot tarkkaan tiedostoa model2.C, jossa funktiomalli min() on määritelty, huomaat, että export-avainsana on ennen mallin määrittelyä. Mallifunktio min() on määritelty *ulkoiseksi* (*exported*) malliksi. Avainsana export ilmaisee kääntäjälle, että mallin määrittelyä ehkä tarvitaan funktiomallien instantioinneissa muissa tiedostoissa. Kääntäjän pitää sitten varmistua, että mallin määrittely on käytettävissä, kun näitä instantiointeja generoidaan.

Esittelemme ulkoisen funktiomallin laittamalla export-avainsanan ennen template-avainsanaa funktion määrittelyssä. Kun funktiomalli on ulkoinen, voimme käyttää mallin instantiointia missä tahansa ohjelman tekstitiedostossa; ainoa asia, joka meidän pitää tehdä, on esitellä malli ennen sen käyttöä. Jos export-avainsana olisi jätetty pois min()-funktiomallin määrittelystä, ei toteutus instantioisi min()-funktiomallista kokonaislukuilmentymää emmekä voisi linkittää ohjelmaamme asianmukaisesti.

Huomaa, että jotkut toteutukset eivät ehkä vaadi export-avainsanaa. Jotkut toteutukset voivat tukea seuraavaa kielen laajennusta: ulkoistamaton funktiomallin määrittely voi esiintyä vain yhdessä ohjelman tekstitiedostossa; muissa ohjelman tekstitiedostoissa käytetyt instantioinnit ovat siitä huolimatta täysin käytettävissä. Tämä piirre on kuitenkin laajennus. C++-standardi vaatii, että käyttäjät merkitsevät funktiomallin määrittelyt ulkoisiksi, jos vain funktiomallin esittely on näkyvissä ohjelmatekstissä ennen mallin instantiointia.

export-avainsanan ei tarvitse esiintyä mallin esittelyssä, joka sijaitsee otsikkotiedostossa. min():in esittelyssä model2.h-tiedostossa ei ole määritelty export-avainsanaa. Avainsana voi olla mukana esittelyssä, mutta se ei ole välttämätöntä.

Funktiomalli pitää määritellä ulkoiseksi malliksi ohjelmassa vain kerran. Koska valitettavasti kääntäjä käsittelee yhtä tiedostoa kerrallaan, se ei tavallisesti havaitse, kun funktiomalli on määritelty useammin kuin kerran ohjelman tekstitiedostossa. Jos sellainen tilanne tulee vastaan, jokin seuraavista saattaa tapahtua:

1. Saattaa generoitua linkitysvirhe siitä, että funktiomalli on määritelty useampaan kuin yhteen tiedostoon.
2. Kääntäjä voi instantioida funktiomallin useammin kuin kerran samoilla malliargumenteilla ja saada siten aikaan linkitysvirheen funktiomallin instantioinnin tuplamäärityksistä.
3. Toteutus voi instantioida funktiomallin käyttäen yhtä ulkoista funktiomäärittelyä ja olla huomioimatta muita määrittelyjä.

Sen vuoksi ei ole varmaa, että virheilmoitus saadaan aikaiseksi, jos ohjelmassamme on useampi kuin yksi ulkoisen funktiomallin määrittely. Pitää huolehtia, että järjestämme ohjelmamme ulkoiset funktiomallien määrittelyt vain yhteen ohjelmatekstitiedostoon.

Erotteleva käännösmalli mahdollistaa, että voimme kätevästi erottaa funktiomalliemme rajapinnat niiden toteutuksista ja voimme järjestää ohjelmamme niin, että funktiomallimme rajapinnat on sijoitettu otsikkotiedostoihin ja niiden toteutukset tekstitiedostoon. Kaikki kääntäjät eivät kuitenkaan tue erottelevaa käännösmallia, ja ne jotka tukevat, eivät aina tue sitä hyvin. Jotta erottelevaa käännösmallia voitaisiin tukea, tarvitaan hienostuneempia ohjelmointiympäristöjä eikä niitä ole kaikissa C++-toteutuksissa. (Oheislukemistossamme, *Inside the C++ Object Model*, kuvataan yhden C++-toteutuksen tukema mallin instantiointimekanismi: Edison Design Group -kääntäjä.)

Tämän kirjan tarkoituksena on, että koska malliemme esimerkit ovat melko pieniä ja koska haluamme niiden kääntyvän helposti monissa C++-toteutuksissa, rajoitumme käyttämään mukaan ottavaa käännösmallia.

10.5.3 Eksplisiittiset instantiointiesittelyt

Kun käytämme mukaan ottavaa käännösmallia, otamme funktiomallien määrittelyt mukaan jokaiseen ohjelmatekstitiedostoon, jossa mallin instantiointia on käytetty. Olemme nähneet, että vaikka emme tarkalleen tiedä, milloin kääntäjä instantioi funktiomallin, pitää ohjelman käyttäytyä *aivan kuin* malli olisi instantioitu tietyllä malliargumenttijoukolla *vain kerran*. Todellisuudessa jotkut kääntäjät (etenkin vanhemmat C++-kääntäjät) instantioivat funktiomallin tietyllä malliargumenttijoukolla useita kertoja. Tässä käännösmallissa valitaan näistä instantioinneista *yksi* instantiointi, jota ohjelma tulee käyttämään (ohjelman linkityksen aikana tai jossain esilinkityksen vaiheessa). Muut instantioinnit jätetään yksinkertaisesti huomiotta.

Onpa funktiomalli instantioitu sitten kerran tai useammin, se ei vaikuta ohjelman tuloksiin, koska ohjelma käyttää loppujen lopuksi vain yhtä mallin instantiointia. Kuitenkin ohjelmamme käännöksenaikaiseen suorituskykyyn voi vaikuttaa, jos funktiomalleja on instantioitu useampia. Jos ohjelma koostuu lukuisista tiedostoista ja jos malli on instantioitu kaikissa näissä tiedostoissa, voi kääntämiseen tarvittava aika kasvaa huomattavasti.

Aikaisempien kääntäjien instantiointiongelmat tekivät mallien käytön hankalaksi. Tämän ratkaisuksi C++-standardi määritteli *eksplisiittisen instantiointiesittelyn*, jolla ohjelmoija ottaa kontrollin siitä, milloin mallien instantioinnit tapahtuvat.

Eksplisiittinen instantiointiesittely on sellainen, jossa template-avainsanan jälkeen tulee funktiomallin instantioinnin esittely, jonka malliargumentit on eksplisiittisesti määritetty. Seuraavassa esimerkissä on tehty eksplisiittinen instantiointiesittely funktiolle sum(int*,int):

```
template <typename Type>
    Type sum( Type op1, int op2 ) { /* ... */ }

// eksplisiittinen instantiointiesittely
template int* sum< int* >( int*, int );
```

Eksplisiittinen instantiointiesittely on pyyntö instantioida malli sum() malliargumentilla int*. Annetun funktiomallin instantioinnin eksplisiittinen instantiointiesittely saa esiintyä vain kerran ohjelmassa.

Funktiomallin määrittelyn pitää olla siinä tiedostossa, jossa eksplisiittinen instantiointiesittely esiintyy. Ellei määrittely ole näkyvissä, eksplisiittinen instantiointiesittely on virheellinen. Esimerkiksi:

```
#include <vector>

template <typename Type>
    Type sum( Type op1, int op2 ); // vain esittely

// määrittele typedef, joka viittaa vektoriin vector< int >
typedef vector< int > VI;

// virhe: sum() määrittelemättä
template VI sum< VI >( VI , int );
```

Kun eksplisiittinen instantiointiesittely esiintyy ohjelmatekstitiedostossa, mitä tapahtuu muissa tiedostoissa, joissa funktiomallin instantiointia on käytetty? Kuinka kerromme kääntäjälle, että eksplisiittinen instantiointiesittely esiintyy toisessa ohjelmatekstitiedostossa eikä funktiomallia pitäisi instantioida, kun sitä käytetään ohjelman muissa tiedostoissa?

Eksplisiittisiä instantiointiesittelyitä käytetään yhdessä kääntäjän valitsimen kanssa, joka vähentää mallien implisiittisiä instantiointeja ohjelmassamme. Tämän valitsimen nimi vaihtelee kääntäjästä toiseen. Esimerkiksi IBM:n kääntäjässä VisualAge for C++ for Windows, versio 3.5, valitsin, joka vähentää implisiittisiä mallien instantiointeja, on nimeltään /ft-. Kun käännämme sovelluksemme tällä valitsimella, kääntäjä olettaa, että käsittelemme mallien instantioinnit eksplisiittisillä instantiointiesittelyillä, eikä se tuolloin instantioi implisiittisesti malleja, joita ohjelmassamme on käytetty.

Tietysti, jos emme käytäkään eksplisiittistä instantiointiesittelyä mallin instantiointiin ja olemme määrittäneet /ft-valitsimen, kun ohjelma käännetään, saamme aikaan linkitysvirheen, joka ilmoittaa, että funktiomallin instantiointi puuttuu. Sellaisissa tapauksissa ei mallia instantioida implisiittisesti.

Harjoitus 10.8

Nimeä kaksi mallin käännösmallia, joita C++ tukee. Selitä, miten ohjelmien funktiomallien määrittelyt on järjestetty kummassakin mallin käännösmallissa.

Harjoitus 10.9

Olkoon määritelty seuraava sum()-malli:

```
template <typename Type>
    Type sum( Type op1, char op2 );
```

Kuinka tekisit eksplisiittisen instantiointiesittelyn malliargumentille, joka on string-tyyppinen?

10.6 Mallin eksplisiittinen erikoistaminen

Ei ole aina mahdollista kirjoittaa yhtä funktiomallia, joka sopii parhaiten kaikkiin mahdollisiin tyyppeihin mallin instantiointia varten. Joissain tapauksissa voimme hyödyntää erityistietämystä tyypistä, jolla voidaan kirjoittaa tehokkaampi funktio kuin sellainen, joka on instantioitu mallista. Toisinaan tavallinen mallin määrittely on yksinkertaisesti väärä tyyppiä varten. Oletetaan esimerkiksi, että on olemassa seuraava määrittely max()-funktiomallia varten:

```
// geneerinen mallin määrittely
template <class T>
    T max( T t1, T t2 ) {
        return (t1 > t2 ? t1 : t2);
}
```

Jos funktiomalli instantioidaan malliargumentilla, joka on tyyppiä const char*, ei geneerinen mallin määrittely toimi oikein, jos haluamme jokaisen argumentin tulkittavan C-tyyliseksi merkkijonoksi eikä osoittimeksi merkkiin. Jotta tämä voitaisiin korjata, pitää tehdä erikoistamismäärittely funktiomallin instantioinnille.

Eksplisiittinen erikoistamismäärittely on sellainen, jossa template-avainsanan jälkeen tulee kulmasulut (< >) ja sen jälkeen funktiomallin erikoistamismäärittely. Tämä määrittely ilmaisee mallin nimen, malliargumentit, joille malli erikoistetaan, funktioparametriluettelon ja funktion rungon. Seuraavassa esimerkissä on tehty eksplisiittinen erikoistamismäärittely funktiolle max(const char*, const char*):

```
#include <cstring>

// const char* eksplisiittinen erikoistaminen:
// korvaa instantioinnin geneerisestä mallin määrittelystä

typedef const char *PCC;
template<> PCC max< PCC >( PCC s1, PCC s2 ) {
    return ( strcmp( s1, s2 ) > 0 ? s1 : s2 );
```

```
}
```

Tämän eksplisiittisen erikoistamisen takia ei mallia instantioida tyypillä const char*, kun funktiota max(const char*, const char*) kutsutaan ohjelmassa. Kaikille kutsuille, jotka kohdistuvat max() -funktioon kahdella const char* -tyyppisellä argumentilla, käynnistetään erikoistamismäärittely; kaikille muille instantiointi suoritetaan geneerisestä mallimäärittelyn instantioinnista ja sitten käynnistetään. Nämä funktiot voidaan käynnistää seuraavasti:

```
#include <iostream>

// funktiomallin max() määrittely
// ja erikoistaminen tyyppiin const char *
// tapahtuu tässä

int main() {
    // kutsu instantiointiin: int max< int >( int, int );
    int i = max( 10, 5 );

    // kutsu eksplisiittiseen erikoistamiseen:
    // const char* max< const char* >( const char*, const char* );
    const char *p = max( "hello", "world" );

    cout << "i: " << i << " p: " << p << endl;
    return 0;
}
```

On mahdollista esitellä funktiomalli eksplisiittisellä erikoistamisella määrittelemättä sitä. Esimerkiksi eksplisiittinen erikoistaminen funktiolle max(const char*, const char*) voidaan esitellä kuten seuraavassa:

```
// funktiomallin eksplisiittisen erikoistamisen esittely
template<> PCC max< PCC >( PCC, PCC );
```

Kun esittelemme tai määrittelemme funktiomallin eksplisiittistä erikoistamista, emme saa jättää pois template-avainsanaa ja sen jälkeen kulmasulkuja < > erikoistamisesittelystä. Samalla tavalla erikoistamisesittelystä ei voi jättää pois funktion parametriluetteloa.

```
// virhe: kelvottomat erikoistamisesittelyt

// malli puuttuu<>
PCC max< PCC >( PCC , PCC );

// funktion parametriluettelo puuttuu
template<> PCC max< PCC >;
```

Malliargumenttien eksplisiittinen erikoistaminen voidaan kuitenkin jättää pois eksplisiittisestä erikoistamisesittelystä, jos malliargumentit voidaan päätellä funktioparametreista.

```
// ok: malliargumentti const char* päätelty parametrityypeistä
template<> PCC max( PCC , PCC );
```

Seuraavassa esimerkissä on funktiomalli sum() eksplisiittisesti erikoistettu:

```
template <class T1, class T2, class T3>
    T1 sum( T2 op1, T3 op2 );

// eksplisiittiset erikoistamisesittelyt

// virhe: T1:n malliargumenttia vi voida päätellä;
//       se pitää määrittää eksplisiittisesti
template<> double sum( float, float );

// ok:T1:n  argumentti eksplisiittisesti määritetty
// T2 ja T3 on päätelty floateiksi
template<> double sum<double>( float, float );

// ok: kaikki argumentit eksplisiittisesti määritetty
template<> int sum<int,char,char>( char, char );
```

template<>-osan jättäminen pois eksplisiittisestä erkioistamisesittelystä ei se aina ole virhe. Esimerkiksi:

```
// geneerisen mallin määrittely
template <class T>
    T max( T t1, T t2 ) { /* ... */ }

// OK: tavallisen funktion esittely
const char* max( const char*, const char* );
```

Tämä max()-funktion esittely ei kuitenkaan muodosta funktiomallille erikoistamista. Sen sijaan se esittelee tavallisen funktion, jolla on paluutyyppi ja parametriluettelo ja joka vastaa mallin instantiointia. Tavallisen funktion esittely, joka vastaa mallin instantiointia, ei ole virhe.

Miksi haluaisimme esitellä tavallisen funktion, joka vastaa mallin instantiointia, sen sijaan, että esittelisimme eksplisiittisen erikoistamisen? Kuten olemme nähneet kohdassa 10.3, vain rajoitettua tyyppikonversiojoukkoa voidaan käyttää funktiomallin instantioinnin argumentin tyypin konvertoimiseksi vastaavaksi funktion parametriksi, jos argumentti ottaa osaa malliargumentin päättelyyn. Näin on myös asianlaita, jos funktiomalli on eksplisiittisesti erikoistettu: vain rajoitettua tyyppikonversiojoukkoa, joka kuvattiin kohdassa 10.3, voidaan käyttää funktiomallin eksplisiittisen erikoistamisen funktioargumentteihin. Eksplisiittiset erikoistamiset eivät auta ohittamaan tyyppikonversioiden rajoituksia. Jos haluamme sallia enemmän kuin rajoitetun tyyppikonversiojoukon, pitää määritellä tavallinen funktio funktiomallin erikoistamisen sijasta. Kohdassa 10.8 tutkitaan tätä yksityiskohtaisemmin ja näytetään, kuinka funktion ylikuormituksen ratkaisu etenee kutsussa, joka täsmää sekä tavallista funktiota että funktiomallin instantiointia.

Funktiomallin eksplisiittinen erikoistaminen voidaan esitellä, vaikka funktiomallia, joka sen erikoistaa, ei ole määritelty. Edellisessä esimerkissä funktiomalli sum() on vain esitelty ennen kuin malli on erikoistettu. Vaikka mallin määrittelyä ei tarvita, mallin esittely tarvitaan. Nimen sum() pitää olla tunnettu mallina, ennen kuin se voidaan erikoistaa.

Funktiomallin eksplisiittisen erikoistamisen esittelyn pitää näkyä ennen kuin sitä käytetään lähdetiedostossa. Esimerkiksi:

```
#include <iostream>
#include <cstring>

// geneerisen mallin määrittely
template <class T>
    T max( T t1, T t2 ) { /* ... */ }

int main() {

    // instantiointi
    // const char* max<const char*>( const char*, const char* );
    // käyttäen geneeristä mallin määrittelyä
    const char *p = max( "hello", "world" );

    cout << " p: " << p << endl;
    return 0;
}

// virhe ohjelmassa: const char* eksplisiittinen erikoistaminen:
// korvaa geneerisen mallin määrittelyn
typedef const char *PCC;
template<> PCC max< PCC >( PCC s1, PCC s2 ) { /* ... */ }
```

Edellisessä esimerkissä käytetään funktion max(const char*,const char*) instantiointia, ennen kuin eksplisiittinen erikoistaminen esitellään. Sen vuoksi kääntäjä on oikeutettu otaksumaan, että funktio pitää instantioida geneerisestä mallin määrittelystä. Ohjelma ei voi kuitenkaan tehdä sekä eksplisiittistä erikoistamista että instantiointia samasta mallista samalla malliargumenttijoukolla. Kun myöhemmin ohjelman tekstitiedostossa kohdataan funktion max(const char*, const char*) eksplisiittinen erikoistaminen, saadaan aikaan käännöksenaikainen virhe.

Jos ohjelma muodostuu useammasta kuin yhdestä tiedostosta, pitää mallin eksplisiittisen erikoistamisen esittelyn olla näkyvissä jokaisessa tiedostossa, jossa erikoistamista käytetään. Funktiomallia ei voida instantioida geneerisestä mallin määrittelystä joissain tiedostoissa ja erikoistaa samoilla malliargumenteilla toisissa tiedostoissa. Mietipä seuraavaa esimerkkiä:

```
// ---- max.h ----
// geneerinen mallin määrittely
template <class Type>
    Type max( Type t1, Type t2 ) { /* ... */ }

// ---- File1.C ----
#include <iostream>
#include "max.h"
void another();

int main() {
```

```
    // instantiointi:
    // const char* max<const char*>( const char*, const char* );
    const char *p = max( "hello", "world" );

    cout << " p: " << p << endl;
    another();

    return 0;
}

// ---- File2.C ----
#include <iostream>
#include <cstring>
# include "max.h"

// mallin eksplisiittinen erikoistaminen tyypille const char*
typedef const char *PCC;
template<> PCC max< PCC >( PCC s1, PCC s2 ) { /*... */ }

void another() {

    // eksplisiittinen erikoistaminen:
    // const char* max< const char* >( const char*, const char* );
    const char *p = max( "hi", "again" );

    cout << " p: " << p << endl;
    return 0;

}
```

Edellinen ohjelma muodostuu kahdesta tiedostosta. Tiedostossa File1.C ei ole esittelyä funktion max(const char*,const char*) eksplisiittiselle erikoistamiselle. Sen sijaan funktiomalli instantioidaan geneerisestä mallin määrittelystä. Tiedostossa File2.C on eksplisiittisen erikoistaminen esittely, ja kutsu max("hi","again") kohdistuu eksplisiittiseen erikoistamiseen. Koska sama ohjelma instantioi funktiomallin max(const char*,const char*) yhdessä tiedostossa ja kutsuu eksplisiittistä erikoistamista toisessa tiedostossa, on tämä ohjelma virheellinen. Jotta tämä ongelma voitaisiin korjata, pitää mallin eksplisiittinen erikoistaminen tehdä ennen kutsua funktioon max(const char*,const char*) File1.C-tiedostossa.

Jotta voisit estää tuollaiset virheet ja varmistua siitä, että funktiomallin max(const char*,const char*) eksplisiittinen erikoistaminen on otettu mukaan jokaiseen tiedostoon, joissa käytetään max()-funktiomallia argumenteilla ja joiden tyyppi on const char*, tulisi eksplisiittisen erikoistamisen esittely sijoittaa otsikkotiedostoon "max.h", joka otetaan mukaan kaikkiin ohjelman tekstitiedostoihin, jotka käyttävät max()-funktiomallia:

```
// ---- max.h ----
// geneerisen mallin määrittely
template <class Type>
    Type max( Type t1, Type t2 ) { /* ... */ }

// mallin eksplisiittisen erikoistamisen esittely tyypille const char*
typedef const char *PCC;
template<> PCC max< PCC >( PCC s1, PCC s2 );

// ---- File1.C ----
#include <iostream>
#include "max.h"
void another();

int main() {

    // erikoistaminen:
    // const char* max<const char*>( const char*, const char* );
    const char *p = max( "hello", "world" );

    //....
}
```

Harjoitus 10.10

Määrittele funktiomalli count(), joka laskee taulukosta jonkin arvon esiintymisen lukumäärän. Kirjoita ohjelma, joka kutsuu sitä. Välitä sille vuorollaan double-, int- ja char-taulukko.

Esittele erikoistettu malli count()-funktion ilmentymästä, joka käsittelee string-merkkijonoja. Aja kirjoittamasi ohjelma uudelleen, jotta se kutsuisi funktiomallien instantiointeja.

10.7 Funktiomallien ylikuormitus ◆A

Funktiomalli voidaan ylikuormittaa. Esimerkiksi seuraavassa ohjelmassa on kolme kelvollista ylikuormitettua esittelyä funktiomallille min():

```
// Array-luokkamallin määrittely
// (esitelty kohdassa 2.4)

template <typename Type>
    class Array{ /* ... */ };

// kolme funktiomallin min() esittelyä

template <typename Type>
    Type min( const Array<Type>&, int ); // #1

template <typename Type>
    Type min( const Type*, int ); // #2

template <typename Type>
    Type min( Type, Type ); // #3
```

Seuraavassa main()-määrittelyssä näytetään, kuinka nämä kolme min()-esittelyä voitaisiin käynnistää:

```
#include <cmath>

int main()
{
    Array<int> iA(1024); // luokan instantiointi
    int ia[1024];

    // Type == int; min( const Array<int>&, int )
    int ival0 = min( iA, 1024 );

    // Type == int; min( const int*, int )
    int ival1 = min( ia, 1024 );

    // Type == double; min( double, double )
    double dval0 = min( sqrt( iA[0] ), sqrt( ia[0] ) );

    return 0;
}
```

Tietysti on niin, että ylikuormitetun funktiomallijoukon esittely ei takaa, että niitä voidaan kutsua onnistuneesti. Ylikuormitetut funktiomallit voivat johtaa moniselitteisyyksiin, kun mallin instantiointia käynnistetään. Tässä on esimerkki sellaisesta moniselitteisyydestä. Näimme aikaisemmin, että seuraavan min5()-mallimäärittelyn

```
template <typename T>
    int min5( T, T ) { /* ... */ }
```

funktiota ei instantioida mallin määrittelystä, jos funktiota min5() kutsutaan erityyppisillä argumenteilla; malliargumentin päättely epäonnistuu ja kutsu on virheellinen, koska T:lle päätellään kaksi erilaista tyyppiä funktion argumenteista.

```
int i;
unsigned int ui;

// ok: tyyppi päätelty T:lle: int
min5( 1024, i );

// malliargumentin päättely epäonnistuu:
// kaksi erilaista tyyppiä päätelty T:lle
min5( i, ui );
```

Jotta toinen kutsu voitaisin ratkaista, voimme ylikuormittaa funktion min5() ja sallia kaksi erilaista argumenttityyppiä:

```
template <typename T, typename U>
    int min5( T, U );
```

Seuraava funktion kutsu käynnistää tämän uuden funktiomallin instantioinnin:

```
// ok: int min5( int, unsigned int )
min5( i, ui );
```

Valitettavasti aikaisempi kutsu on nyt moniselitteinen:

```
// virhe: moniselitteinen: kaksi mahdollista instantiointia
//        from min5( T, T ) and min5( T, U )
min5( 1024, i );
```

min5()-funktion toinen esittely sallii kaksi eri tyyppiä funktioargumenteiksi. Se ei kuitenkaan tarkoita, että niiden pitää olla erilaiset. T ja U voivat molemmat olla int-tyyppisiä tässä tapauksessa. Molemmat malliesittelyt voidaan instantioida kutsulla, jossa kahdella funktioargumentilla on sama tyyppi. Ainoa tapa, jolla ilmaistaan, kumpi funktiomalli on parempi ja saadaan kutsu yksiselitteiseksi, on määrittää malliargumentit eksplisiittisesti (katso kohdasta 10.4 eksplisiittisten malliargumenttien käsittely). Esimerkiksi:

```
// OK: instantiointi: min5( T, U )
min5<int, int>( 1024, i );
```

Kuitenkin tässä tapauksessa voimme hävittää funktiomallin ylikuormittamisen kokonaan. Koska min5(T,U):n käsittelemä kutsujoukko on niiden kutsujen ylijoukko (*superset*), jonka min5(T,T) käsittelee, tarvitaan vain ainoastaan esittely min5(T,U), jolloin min5(T,T) voidaan poistaa. Siitä syystä, kuten kerroimme luvun 9 alussa, vaikka ylikuormitus on mahdollista, pitää olla huolellinen, kun suunnitellaan ylikuormitettuja funktioita ja varmistua, että ylikuormitus on välttämätöntä. Nämä suunnittelurajoitukset pätevät myös silloin, kun suunnittelemme ylikuormitettuja funktiomalleja.

Joissakin tilanteissa funktion kutsu voi olla moniselitteinen, vaikka kaksi erilaista funktiomallia voidaan instantioida kutsua varten. Kun on tehty seuraavat kaksi mallimäärittelyä funktiosta sum(), on tässä tilanne, jossa ensimmäinen mallimäärittely on parempi, vaikka instantiointi voidaan generoida kummasta tahansa funktiomallista.

```
template <typename Type>
    Type sum( Type*, int );

template <typename Type>
    Type sum( Type, int );

int ia[1024];

// Type == int ; sum<int>( int*, int );   tai
// Type == int*; sum<int*>( int*, int );  ??
int ival1 = sum<int>( ia, 1024 );
```

Yllättävää kyllä, edellinen kutsu ei ole moniselitteinen. Malli instantioidaan käyttämällä ensimmäistä mallimäärittelyä. Mallimäärittely, joka on *erikoistunein*, valitaan instantiointia varten. Type:n malliargumentti on siten int eikä int*.

Jotta yksi malli olisi erikoistuneempi kuin toinen, pitää molemmilla malleilla olla sama nimi, sama määrä parametreja. Lisäksi funktioparametreista, jotka ovat erityyppisiä kuten T* ja T aikaisemmin, yhden pitää kyetä hyväksymään sen argumentin ylijoukon, jonka vastaava parametri toisessa mallissa voi ottaa vastaan. Esimerkiksi mallissa sum(Type*, int) voi ensimmäinen funktioparametri täsmätä vain osoitintyyppisten argumenttien kanssa. Mallissa sum(Type, int) ensimmäinen funktioparametri voi täsmätä argumenttien kanssa, jotka ovat osoitintyyppisiä tai yhtä hyvin mitä tahansa tyyppejä. Toinen malli hyväksyy ensimmäisen mallin hyväksymän tyyppien ylijoukon. Mallin, joka hyväksyy rajoitetumman argumenttijoukon, sanotaan olevan erikoistuneempi. Esimerkissämme malli sum(Type*, int) on erikoistuneempi, ja se instantioidaan kutsua varten.

10.8 Instantiointien ylikuormituksen ratkaisu

Kuten näimme edellisessä kohdassa, funktiomalli voidaan ylikuormittaa. Funktiomallilla voi olla sama nimi kuin tavallisella mallittomalla funktiolla. Esimerkiksi:

```
// funktiomalli
template <class Type>
    Type sum( Type, int ) { /* ... */ }

// tavallinen (malliton) funktio
double sum( double, double );
```

Kun ohjelma kutsuu sum()-funktiota, kutsu voidaan ratkaista joko funktiomallin instantioinniksi tai tavallisen funktion kutsuksi. Se, kumpaa kutsutaan, riippuu siitä, kumpi näistä funktioista parhaiten vastaa funktion argumenttien tyyppejä. Luvussa 9 esiteltyä funktion ylikuormituksen ratkaisun prosessia käytetään päättelyyn, mikä funktioista parhaiten vastaa funktion kutsun argumentteja. Mietipä esimerkiksi seuraavaa:

```
void calc( int ii, double dd ) {
    // kutsuuko se mallin instantiointia
    // vai tavallista funktiota?
    sum( dd, ii );
}
```

Kutsuuko sum(dd, ii) funktiota, joka instantioidaan mallista, vai kutsuuko se tavallista funktiota? Jotta voisimme vastata tähän kysymykseen, käykäämme läpi funktion ylikuormituksen ratkaisun prosessi. Funktion ylikuormituksen ratkaisun ensimmäinen vaihe on hakea niiden funktioehdokkaitten joukko, joita voidaan kutsua. Tämä joukko muodostuu funktioista, joilla on sama nimi kuin kutsussa ja joiden esittelyt näkyvät kutsupaikkaan.

Kun funktiomalli on olemassa, tuon mallin instantiointi on funktioehdokas, jos funktio voidaan instantioida käyttämällä funktion kutsun argumentteja. Voidaanko funktio instantioida, riippuu siitä, onnistuuko malliargumenttien päättely. (Malliargumenttien päättelyprosessi on kerrottu kohdassa 10.3.) Edellisessä esimerkissä käytetään funktioargumenttia dd malliargumentin päättelyyn Type:lle. Päätelty malliargumentti on double ja mallin instantiointi sum(double, int) lisätään funktioehdokkaitten joukkoon. Siten on olemassa kaksi funktioehdokasta kutsua varten: mallin instantiointi sum(double,int) ja tavallinen funktio sum(double, double).

Kun mallien instantioinnit on lisätty funktioehdokkaitten joukkoon, voi funktion ylikuormituksen ratkaisu edetä kuten tavallisesti.

Funktion ylikuormituksen ratkaisun toinen vaihe valitsee elinkelpoisten funktioiden joukon funktioehdokkaista. Muista, että elinkelpoinen funktio on sellainen funktioehdokas, jolle on olemassa tyyppikonversioita, joilla jokainen funktioargumentti voidaan konvertoida vastaavaksi funktioparametriksi. (Kohdassa 9.3 on esitelty tyyppikonversiot, joita voidaan käyttää funktion argumentteihin.) Tyyppikonversiot ovat olemassa sekä instantioinnille sum(double,int) että mallittomalle funktiolle sum(double, double). Nämä ovat molemmat elinkelpoisia

funktioita.

Funktion ylikuormituksen ratkaisun kolmas vaihe laittaa argumenttien tyyppikonversiot paremmuusjärjestyksen, jotta voitaisiin valita parhaiten elinkelpoinen funktio. Meidän esimerkissämme järjestys on seuraava:

- Funktiomallin instantioinnille sum(double,int):

1. Ensimmäisen argumentin ja parametrin tyyppi on double, jolloin konversio vastaa täydellisesti.
2. Toisen argumentin ja parametrin tyyppi on int, jolloin konversio vastaa myös täydellisesti.

- Mallittomalle funktiolle sum(double,double):

1. Ensimmäisen argumentin ja parametrin tyyppi on double, jolloin konversio vastaa täydellisesti.
2. Toisen argumentin tyyppi on int ja parametrin tyyppi on double; käytetty konversio on liukuluku-kokonaisvakiokonversio.

Molemmat funktiot ovat yhtä hyviä, kun verrataan ensimmäistä argumenttia. Kuitenkin funktiomallin instantiointi on parempi toisen argumentin takia. Siten parhaiten elinkelpoiseksi funktioksi valittu funktio kutsua varten on instantiointi sum(double,int).

Funktiomallin instantiointi pääsee funktioehdokkaitten joukkoon vain, jos malliargumentin päättely onnistuu. Ei ole virhe, jos malliargumentin päättely epäonnistuu; sellaisessa tapauksessa ei funktion instantiointia lisätä funktioehdokkaitten joukkoon. Oletetaan esimerkiksi, että funktiomalli sum() on esitelty seuraavasti:

```
// funktiomalli
template <class T>
    int sum( T*, int ) { .... }
```

Kun käytämme samaa funktion kutsua kuten aikaisemmin, malliargumentin päättely epäonnistuu, koska funktion argumenttia, joka on tyyppiä double, ei voi saada vastaavaksi parametrityypiksi T*. Koska funktion instantiointia ei voida generoida funktiomallista tätä kutsua varten, ei instantiointia lisätä funktioehdokkaitten joukkoon. Ainoa funktioehdokkaitten joukossa oleva funktio on sum(double,double). Tuo funktio valitaan kutsua varten ja toinen argumentti konvertoidaan double-tyypiksi.

Mitä sitten, jos malliargumentin päättely onnistuu, mutta malli on eksplisiittisesti erikoistettu päätellyille malliargumenteille? Silloin lisätään eksplisiittinen erikoistaminen funktioehdokkaitten joukkoon funktion sijasta, joka instantioitaisiin geneerisestä mallimäärittelystä. Esimerkiksi:

```
// funktiomallin määrittely
template <class Type> Type sum( Type, int ) { /* ... */ }

// eksplisiittinen erikoistaminen: Type == double
template<> double sum<double>( double, int );
```

```
// tavallinen (malliton) funktio
double sum( double, double );

void manip( int ii, double dd ) {
    // kutsuu mallin eksplisiittistä erikoistamista sum<double>()
    sum( dd, ii );
}
```

Kun funktiossa manip() kutsutaan funktiota sum(), huomaa malliargumentin päättely, että geneerisestä mallimäärittelystä instantioitu sum(double,int) tulisi lisätä funktioehdokkaitten joukkoon. Kuitenkin kutsulle sum(double,int) on olemassa eksplisiittinen erikoistaminen, jolloin tämä lisätään funktioehdokkaitten joukkoon. Itse asiassa, koska tämän erikoistamisen huomataan myöhemmin parhaiten sopivan kutsua varten, tämä funktio on se, jonka funktion ylikuormituksen ratkaisu valitsee.

Mallien eksplisiittisiä erikoistamisia ei lisätä automaattisesti funktioehdokkaitten joukkoon. Vain, jos malliargumentin päättely onnistuu, mallin eksplisiittistä erikoistamista harkitaan funktion kutsua varten. Esimerkiksi:

```
// funktiomallin määrittely
template <class Type>
    Type min( Type, Type ) { /* ... */ }

// eksplisiittinen erikoistaminen: Type == double
template<> double min<double>( double, double );

void manip( int ii, double dd ) {
    // virhe: malliargumentin päättely epäonnistuu
    //      ei funktioehdokasta kutsua varten
    min( dd, ii );
}
```

Funktiomalli min() on erikoistettu malliargumentille double. Tätä erikoistamista ei kuitenkaan lisätä funktioehdokkaitten joukkoon. Malliargumentin päättely epäonnistuu, kun funktiossa manip() kutsutaan funktiota min(), koska jokaisesta funktioargumentista Type:lle päätelty malliargumentti on erilainen. Ensimmäisestä argumentista päätellään Type:lle tyyppi double. Toisesta argumentista päätellään Type:lle tyyppi int. Koska malliargumentin päättely epäonnistuu, ei instantiointia lisätä funktioehdokkaitten joukkoon ja min(double,double)-erikoistaminen jätetään huomiotta. Koska muita funktioehdokkaita ei ole kutsua varten, kutsu on virheellinen.

Kuten mainittiin kohdassa 10.6, tavallisella funktiolla voi olla paluutyyppi ja parametriluettelo, jotka vastaavat täydellisesti sitä funktiota, joka voitaisiin instantioida mallista. Seuraavassa esimerkissä funktio min(int,int) on tavallinen funktio eikä sitä ole erikoistettu funktiomallille min(), koska kuten muistat, erikoistamisen esittely pitää alkaa ilmaisulla template<>:

```
// funktiomallin esittely
template <class T>
    T min( T, T );

// tavallinen funktio min( int, int )
int min( int, int ) { }
```

Funktion kutsu voisi täsmätä yhtä hyvin tähän tavalliseen funktioon kuin funktiomallista instantioituun funktioon. Seuraavassa esimerkissä kutsun min(ai[0],99) molemmat argumentit ovat int-tyyppisiä. Tätä kutsua varten on olemassa kaksi elinkelpoista funktiota: tavallinen funktio min(int,int) ja funktio, jolla on sama paluutyyppi ja parametriluettelo ja joka on instantioitu funktiomallista.

```
int ai[4] = { 22, 33, 44, 55 };
int main() {
    // kutsuu tavallista funktiota min( int, int )
    min( ai[0], 99 );
}
```

Sellainen kutsu ei kuitenkaan ole moniselitteinen. Kun malliton funktio on mukana, annetaan sille etuoikeus, koska se on eksplisiittisesti toteutettu. Funktion ylikuormituksen ratkaisu valitsee tavallisen funktion min(int,int) kutsua varten.

Kun funktion ylikuormitus ratkaisee kutsuvansa tavallista funktiota, ei myöhemmin ole paluuta, ellei ohjelma sisällä tämän funktion määrittelyä. Funktiomallia ei instantioida eikä funktion runkoa luoda, ellei funktion määrittelyä löydy. Sen sijaan siitä aiheutuu linkityksenaikainen virhe. Seuraavassa esimerkissä ohjelma kutsuu, mutta ei määrittele tavallista funktiota min(int,int). Tämä ohjelma generoi linkitysvirheen:

```
// funktiomalli
template <class T>
    T min( T, T ) { .... }

// tätä tavallista funktiota ei ole määritelty tässä ohjelmassa
int min( int, int );

int ai[4] = { 22, 33, 44, 55 };
int main() {
    // linkitysvirhe: min( int, int ) kutsuttu, mutta ei määritelty
    min( ai[0], 99 );
}
```

Miksi olisi hyödyllistä määritellä tavallinen funktio, jonka paluutyyppi ja parametriluettelo vastaisivat mallista instantioidun funktion paluutyyppiä ja parametriluetteloa? Muista, että kun kutsumme funktiota, joka on instantioitu mallista, voimme käyttää funktion argumenttiin vain rajoitettua tyyppikonversiojoukkoa, jota käytetään malliargumentin päättelyyn. Jos esittelemme tavallisen funktion, kaikkia tyyppikonversioita harkitaan argumenttien konvertoimiseksi, koska tavallisen funktion parametrit ovat kiinteitä. Katsokaamme esimerkkiä, josta

nähdään, miksi haluaisimme esitellä tavallisen funktion.

Oletetaan, että haluamme määritellä funktiomallin erikoistamisen min <int>(int,int) ja että tämä funktio käynnistetään, kun funktiota min() kutsutaan millä tahansa kokonaislukutyyppisillä argumenteilla, olivatpa argumentit samantyyppisiä tai eivät. Tyyppikonversioiden rajoitusten takia kutsut, joissa argumentit ovat erityyppisiä kokonaislukuja, eivät käynnistä funktiomallin instantiointia min<int>(int,int) suoraan. Voisimme kutsua instantiointia suoraan määrittämällä eksplisiittiset malliargumentit. Pidämme kuitenkin parempana ratkaisua, jossa ei vaadita joka kutsupaikan muokkausta. Kun määrittelemme tavallisen funktion, ohjelmamme käynnistää erikoistetun versiomme min(int,int) aina, kun kokonaislukutyyppisiä argumentteja käytetään ilman tarvetta käyttää eksplisiittisiä malliargumentteja jokaisessa funktion kutsussa. Esimerkiksi:

```
// funktiomallin määrittely
template <class Type>
    Type min( Type t1, Type t2 ) { ... }

int ai[4] = { 22, 33, 44, 55 };
short ss = 88;

void call_instantiation() {
    // virhe: ei funktioehdokasta tälle kutsulle
    min( ai[0], ss );
}

// tavallinen funktio
int min( int a1, int a2 ) {
    min<int>( a1, a2 );
}

int main() {
    call_instantiation();
    // kutsuu tavallista funktiota
    min( ai[0], ss );
}
```

Funktiossa call_instantiation() ei ole kutsulle min(ai[0],ss) funktioehdokkaita. Yritys generoida funktioehdokas funktiomallista min() epäonnistuu, koska Type:lle päätellään erilaiset malliargumentit funktioargumenteista. Kutsu on siksi virheellinen. Kuitenkin main()-funktiossa on kutsulle min(ai[0],ss) tavallisen min(int,int)-funktion esittely näkyvissä. Tämä tavallinen funktio on elinkelpoinen kutsua varten: ensimmäisen argumentin tyyppi täsmää täysin sitä vastaavaan parametriin ja toinen argumentti voidaan konvertoida sitä vastaavan parametrin tyypiksi ylennyksen avulla. Tämä tavallinen funktio on ainoa elinkelpoinen funktio ja se valitaan toista kutsua varten.

Nyt, kun olemme näyttäneet, kuinka funktion ylikuormituksen ratkaisu etenee, kun mukana on funktiomallien instantiointeja, funktiomallien erikoistamisia ja samannimisiä funktioita,

tehkäämme yhteenveto funktion ylikuormituksen ratkaisun vaiheista kutsua varten, jossa harkitaan tavallisia funktioita ja funktiomalleja keskenään:

1. Muodosta funktioehdokkaitten joukko.

   Ne funktiomallit otetaan huomioon, joilla on sama nimi kuin kutsutulla funktiolla. Jos malliargumentin päättely onnistuu funktion kutsun argumenteille, funktiomalli instantioidaan tai, jos on olemassa mallin erikoistaminen päätellylle malliargumentille, mallin erikoistaminen on funktioehdokas.
2. Muodosta elinkelpoisten funktioiden joukko kuten kohdassa 9.3 kuvattiin.

   Pidä funktioehdokkaista vain ne, joita voidaan kutsua funktion kutsun argumenteilla.
3. Laita tyyppikonversiot paremmuusjärjestykseen kuten kuvattiin kohdassa 9.3.
   a. Jos vain yksi funktio tulee valituksi, kutsu tuota funktiota.
   b. Jos kutsu on moniselitteinen, poista funktiomallin instantioinnit elinkelpoisten funktioiden joukosta.
4. Suorita ylikuormituksen ratkaisu ottamalla huomioon vain tavalliset funktiot elinkelpoisten funktioiden joukosta kuten kohdassa 9.3 kuvattiin.
   a. Jos vain yksi funktio tulee valituksi, kutsu tuota funktiota.
   b. Muussa tapauksessa kutsu on moniselitteinen.

Käykäämme läpi esimerkki. Tässä on kaksi esittelyä: funktiomallin esittely ja tavallinen funktio, joka saa kaksi double-tyyppistä argumenttia.

```
template <class Type>
    Type max( Type, Type ) { .... }

// tavallinen funktio
double max( double, double );
```

Seuraavassa on kolme kutsua max()-funktioon. Osaatko kertoa, mikä ilmentymä käynnistetään kutsua varten?

```
int main() {
    int ival;
    double dval;
    float fd;

    // jotkin arvot sijoitetaan muuttujiin ival, dval ja fd

    max( 0, ival );
    max( 0.25, dval );
    max( 0, fd );
}
```

Katsokaamme jokaista kutsua vuorollaan.

- max(0,ival): Molemmat argumentit ovat int-tyyppisiä. Tätä kutsua varten on kaksi funktioehdokasta: funktiomallin instantiointi max(int,int) ja tavallinen funktio max(double,double). Funktiomallin instantiointi täsmää täysin funktion argumentteihin. Tämä funktio käynnistetään.
- max(0.25,dval): Molemmat argumentit ovat double-tyyppisiä. Kutsua varten on kaksi funktioehdokasta: funktiomallin instantiointi max(double,double) ja tavallinen funktio max(double,double). Kutsu on siksi moniselitteinen, koska se täsmää täysin molempia funktioita. Sääntö 3b ilmaisee, että tavallinen funktio valitaan tässä tapauksessa.
- max(0,fd): Argumentit ovat vastaavasti tyypit int ja float. Tätä kutsua varten on vain yksi funktioehdokas: tavallinen funktio max(double,double). Malliargumentin päättely epäonnistuu, koska kahdesta funktioargumentista Type:lle päätelty malliargumentti ei ole yksi tyyppi; siksi mallin instantiointia ei lisätä funktioehdokkaitten joukkoon. Tavallinen funktio on elinkelpoinen funktio, koska on olemassa tyyppikonversioita, joilla argumentit voidaan konvertoida vastaaviksi funktion parametrien tyypeiksi. Tavallinen funktio tulee valituksi. Ellei tavallista funktiota olisi esitelty, kutsu olisi ollut virheellinen.

Mitä jos olisimme määritelleet toisen tavallisen max()-funktion? Esimerkiksi:

```
template <class T> T max( T, T ) { .... }

// kaksi tavallista funktiota
char max( char, char );
double max( double, double );
```

Onko kolmannen kutsun ratkaisu yhtään erilainen? Kyllä on.

```
int main() {
    float fd;

    // mihin funktioon ratkaistu?
    max( 0, fd );
}
```

Sääntö 3b ilmoittaa, että koska kutsu on moniselitteinen, tavalliset funktiot huomioidaan. Kumpaakaan näistä funktioista ei valita elinkelpoiseksi funktioksi, koska argumenteille tehtävät tyyppikonversiot ovat yhtä huonoja molemmille funktioille: molemmat argumentit vaativat vakiokonversion, jotta ne täsmäisivät kummankaan elinkelpoisen funktion vastaaviin parametreihin. Kutsu on siksi moniselitteinen ja aiheuttaa käännöksenaikaisen virheen.

Harjoitus 10.11

Palatkaamme aikaisemmin esitettyyn esimerkkiin:

```
template <class Type>
```

```
    Type max( Type, Type ) { .... }

double max( double, double );

int main() {
    int ival;
    double dval;
    float fd;

    max( 0, ival );
    max( 0.25, dval );
    max( 0, fd );
}
```

Seuraava funktiomallin erikoistaminen on lisätty esittelyjen joukkoon globaalille viittausalueelle:

```
template <> char max<char>( char, char ) { .... }
```

Käy uudelleen läpi main()-funktion kutsut ja luettele funktioehdokkaat sekä elinkelpoiset funktiot jokaiselle kutsulle.

Oletetaan, että seuraava funktion kutsu on lisätty main()-funktioon. Mihin funktioon kutsu ratkaistaan? Miksi?

```
int main() {
    // ...
    max (0, 'J' );
}
```

Harjoitus 10.12

Oletetaan, että on olemassa seuraavat mallimäärittelyt, erikoistamiset, muuttujien ja funktioiden esittelyt:

```
int i;          unsigned int ui;
char str[24];   int ia[24];

template <class T> T calc( T*, int );
template <class T> T calc( T, T );
template<> char calc( char*, int );
double calc( double, double );
```

Yksilöi, mitkä mallin instantioinneista tai funktioista käynnistetään jokaista seuraavaa kutsua varten. Luettele jokaisesta kutsusta funktioehdokkaat ja elinkelpoiset funktiot ja perustele, miksi elinkelpoisin funktio tulee valituksi.

```
(a) calc( str, 24 );  (d) calc( i, ui );
(b) calc( ia, 24 );   (e) calc( ia, ui );
(c) calc( ia[0], i ); (f) calc( &i, i );
```

10.9 Nimiresoluutio mallimäärittelyissä ◆

Mallimäärittelyn sisällä joidenkin rakenteiden merkitys eroaa mallin instantioinnista toiseen, kun taas joidenkin rakenteiden merkitys on sama mallin kaikissa instantioinneissa. Tämä riippuu siitä, kuuluuko rakenteeseen malliparametria. Esimerkiksi:

```
template<typename Type>
    Type min( Type* array, int size )
{
    Type min_val = array[0];
    for ( int i = 1; i < size; ++i )
        if ( array[i] < min_val )
            min_val = array[i];

    print( "Minimum value found: " );
    print( min_val );

    return min_val;
}
```

min()-funktiossa array:n ja min_val:in tyypit riippuvat todellisista tyypeistä, joilla Type korvataan mallin instantioinnissa, kun taas esimerkiksi size:n tyyppi säilyy malliparametrin tyypistä huolimatta. array:n ja min_val:in tyypit vaihtelevat mallin instantioinnista toiseen. Tämän vuoksi sanomme, että näiden muuttujien tyypit *riippuvat malliparametrista*, kun taas size:n tyyppi ei riipu malliparametrista.

Koska min_val:in tyyppi on tuntematon, myös se operaatio on tuntematon, jota käytetään, kun min_val esiintyy lausekkeessa. Mitä print()-funktiota esimerkiksi tulisi funktion print(min_val) kutsua? Tulisiko sen olla print()-funktio, jota käytetään int-tyypeille? Vai tulisiko sen olla funktio float-tyypeille? Onko kutsu virheellinen, koska ei ole print()-funktiota, jota voitaisiin kutsua min_val:in tyyppisellä argumentilla? On mahdotonta vastata näihin kysymyksiin ennen kuin malli on instantioitu ja kunnes tiedämme, minkä tyyppinen min_val on. Tämän vuoksi sanomme myös, että kutsu print(min_val) riippuu malliparametrista.

Ei ole olemassa sellaisia asioita min()-funktion rakenteista, jotka eivät riipu malliparametrista. Aina tiedetään esimerkiksi, mitä funktiota tulisi käyttää kutsussa print("Minimum value found: "). Se on funktio, jota käytetään merkkijonojen tulostamiseen. Kutsuttu print()-funktio ei vaihtele mallin instantioinnista toiseen. Siksi sanomme, että tämä kutsu ei riipu malliparametrista.

Kuten opimme luvussa 7, C++:ssa funktio pitää esitellä ennen kuin sitä voidaan kutsua. Pitääkö mallimäärittelyssä kutsuttu funktio esitellä ennen kuin mallimäärittely näkyy? Pitääkö edellisessä esimerkissä print()-funktio esitellä ennen kuin kutsu näkyy min():in mallimäärittelyssä? Vastaus riippuu nimestä, johon viittaamme. Rakenne, joka ei riipu malliparametrista, pitää esitellä ennen kuin sitä käytetään mallimäärittelyssä. Funktiomallin min() määrittely on siten virheellinen kuten aikaisemmin esitettiin. Koska kutsu

```
print( "Minimum value found: " );
```

ei ole sellainen, joka riippuu malliparametrista, pitää print()-funktio merkkijonoille esitellä ennen kuin sitä voidaan käyttää mallimäärittelyssä. Tämän ongelman ratkaisemiseksi print()-funktion esittely voidaan laittaa ennen min()-funktion määrittelyä kuten seuraavassa:

```
// ---- primer.h ----

// tämä esittely on välttämätön:
// funktiota print( const char * ) kutsutaan min()-funktiossa
void print( const char* );

template<typename Type>
    Type min( Type* array, int size ) {

    // ....

    print( "Minimum value found: " );
    print( min_val );

    return min_val;
}
```

Toisaalta print()-funktion esittely, jota print(min_val)-kutsussa käytetään, ei vielä ole se, mitä tarvitaan, koska emme vielä tiedä, mitä print()-funktiota etsitään. Ei ole mahdollista tietää, mitä print()-funktiota print(min_val) kutsuu ennen kuin min_val:in tyyppi on tunnettu.

Joten, pitääkö kutsun print(min_val) käynnistämä print()-funktio esitellä? Se pitää esitellä ennen kuin malli instantioidaan. Esimerkiksi:

```
#include <primer.h>
void print( int );

int ai[4] = { 12, 8, 73, 45 };

int main() {
    int size = sizeof(ai) / sizeof(int);
    // instantiointi: min( int*, int )
    min( &ai[0], size );
}
```

main()-funktio kutsuu funktiomallin instantiointia min(int*,int). Tässä min():in instantioinnissa Type korvataan int-tyypillä ja muuttujan min_val tyyppi on int. Siksi funktion kutsu print(min_val) kohdistuu funktioon, joka voidaan käynnistää int-tyyppisellä argumentilla. Silloin, kun min(int*,int) instantioidaan, tiedämme, että print()-funktion toisessa kutsussa on int-tyyppinen argumentti. Tämä on se hetki, kun print()-funktiota kutsutaan int-tyyppisellä argumentilla ja funktion pitää olla näkyvissä. Esimerkissämme valittu funktio on print(int). Ellei print(int)-funktiota ole esitelty ennen kuin min(int*,int)-instantiointi tapahtuu, instantiointi saa aikaan käännöksenaikaisen virheen.

Tästä syystä funktiomallin nimiresoluutio tapahtuu kahdessa vaiheessa: ensiksi ratkaistaan

nimet, jotka eivät riipu malliparametrista, kun malli määritellään; toiseksi ratkaistaan nimet, jotka riippuvat malliparametrista, kun malli instantioidaan. Voit ihmetellä, miksi on nämä kaksi vaihetta. Miksi esimerkiksi ei kaikkia nimiä ratkaista, kun malli instantioidaan?

Jos olisimme funktiomallin suunnittelijoita, haluaisimme kontrolloida, kuinka mallimäärittelyn nimet ratkaistaan. Olettakaamme, että funktiomalli min() on osa kirjastoa, jossa määritellään muitakin malleja ja funktioita. Haluamme min():in toteutuksen käyttävän kirjastomme muita komponentteja aina, kun se on mahdollista. Edellisessä esimerkissä kirjastomme rajapinta on määritelty otsikkotiedostoon <primer.h>. Sekä funktion print(const char*) esittely että funktiomallin min() määrittely ovat osa kirjastomme rajapintaa. Haluamme min():in instantiointien kutsuvan kirjastomme print()-funktiota. Nimiresoluution ensimmäinen vaihe takaa, että tämä tapahtuu. Kun nimi, jota on käytetty mallimäärittelyssä, ei riipu malliparametrista, on taattu, että nimi viittaa kirjastomme toiseen komponenttiin — tämä tarkoittaa, että se viittaa esittelyyn, jonka paketoimme funktiomallin määrittelyn kanssa <primer.h>-tiedostoon.

Itse asiassa funktiomallin suunnittelijan pitää varmistua, että tulee esiteltyä kaikki mallimäärittelyssä käytetyt nimet, jotka eivät riipu malliparametrista. Jos mallimäärittelyssä käytetty nimi ei riipu malliparametrista ja jos tämän nimen esittelyä ei löydy, kun malli määritellään, mallimäärittely on virheellinen. Virhettä ei havaita koskaan silloin, kun malli instantioidaan. Esimerkiksi:

```
// ---- primer.h ----
template<typename Type>
    Type min( Type* array, int size )
{
    Type min_val = array[0];
    // ...
    // virhe: print ( const char* ) ei löydy
    print( "Minimum value found: " );

    // ok: riippuu malliparametrista
    print( min_val );
    // ...
}

// ---- user.C ----
#include <primer.h>

// tämä print( const char* )-esittely jätetään huomiotta
void print( const char* );
void print( int );

int ai[4] = { 12, 8, 73, 45 };

int main() {
    int size = sizeof(ai) / sizeof(int);
    // instantiointi: min( int*, int )
    min( &ai[0], size );
```

```
}
```

Tiedoston user.C esittely print(const char*) ei ole näkyvissä siellä, missä mallimäärittely esiintyy. Kuitenkin sellainen esittely on näkyvissä siellä, missä malli min-(int*,int) instantioidaan, mutta tätä esittelyä ei oteta huomioon kutsussa print("Minimum value found: "), koska kutsu ei riipu malliparametrista. Ellei mallimäärittelyssä oleva rakenne riipu malliparametrista, nimet ratkaistaan mallimäärittelyn yhteydessä, eikä tätä ratkaisua harkita uudelleen koskaan mallin instantioinnin yhteydessä. Tästä syystä mallin suunnittelijan vastuulla on varmistua, että mallimäärittelyssä käytettyjen nimien esittelyt on kunnolla otettu mukaan mallimäärittelyyn osana kirjaston rajapintaa.

Vaihdetaanpa nyt hattuja ja oletetaan, että kirjaston oli kirjoittanut joku muu, ja että olemme sen sijaan <primer.h>-otsikkotiedostossa määritellyn kirjaston käyttäjiä. On olemassa tilanteita, joissa haluamme ohjelmaamme määritellyt oliot ja funktiot otettavan huomioon, kun ohjelmamme instantioi mallin kirjastosta. Olettakaamme esimerkiksi, että ohjelmassamme on määritelty luokkatyyppi nimeltään SmallInt. Haluamme instantioida funktion min() <primer.h>-kirjastosta saadaksemme minimiarvon taulukosta, jonka oliot ovat SmallInt-tyyppisiä.

Kun malli min() instantioidaan taulukon SmallInt-tyyppisiä olioita varten, Type:n malliargumentti on SmallInt-luokkatyyppi. Tämä tarkoittaa, että min_val:in tyyppi min():in instantioinnissa on SmallInt. Minkä print()-funktion tulisi ratkaista min():in instantioinnissa kutsu print(min_val) ?

```
// ---- user.h ----
class SmallInt { /* ... */ };
void print( const SmallInt & );

// ---- user.C ----
#include <primer.h>
#include "user.h"
SmallInt asi[4];

int main() {
    // aseta asi:n elementit

    // instantiointi: min( SmallInt*, int )
    int size = sizeof(asi) / sizeof(SmallInt);
    min( &asi[0], size );
}
```

Aivain oikein — haluamme funktiomme print(const SmallInt &) tulevan huomioonotetuksi. Vain <primer.h>-kirjastoon määriteltyjen funktioiden huomioon ottaminen ei aivan riitä. Nimiresoluution toinen vaihe takaa, että tämä tapahtuu. Kun mallimäärittelyssä käytetty nimi riippuu malliparametrista, otetaan instantioinnin yhteydessä esitellyt nimet huomioon. Siitä syystä olemme varmoja, että funktiot, jotka käsittelevät SmallInt-tyyppisiä olioita, otetaan huomioon funktiomallin operaatioissa, jos malliargumentin tyyppi on SmallInt.

Paikkaa, jossa malli instantioidaan lähdekoodissa, nimitetään mallin *instantiointipaikaksi*. Mallin instantiointipaikan tietäminen on tärkeää, koska se määrää, mitkä esittelyt otetaan huo-

mioon nimille, jotka riippuvat malliparametrista. Funktiomallin instantiointipaikka on aina nimiavaruuden viittausalueella ja sen jälkeen on aina funktio, joka viittaa instantiointiin. Esimerkiksi min(SmallInt*,int):in instantiointipaikka sijaitsee heti main()-funktion jälkeen nimiavaruuden viittausalueella:

```
// ...
int main() {
    // ...
    // min( SmallInt*, int )-funktion käyttö
    min( &asi[0], size );
}
// min( SmallInt*, int ):in instantiointipaikka
// aivan kuin instantioinnin määrittely esiintyisi kuten seuraavassa:
SmallInt min( SmallInt* array, int size )
    { /* ... */ }
```

Mutta mitä jos mallin instantiointia on käytetty useammin kuin kerran lähdekooditiedostossa? Missä on instantiointipaikka? Miksi sillä on merkitystä, voit kysyä. SmallInt-esimerkissämme sillä on merkitystä, koska print(const SmallInt &)-funktiomme esittelyn pitää esiintyä ennen min(SmallInt*,int):in instantiointipaikkaa. Esimerkiksi:

```
#include <primer.h>
void another();

SmallInt asi[4];

int main() {
    // aseta asi:n elementit
    int size = sizeof(asi) / sizeof(SmallInt);
    min( &asi[0], size );

    another();
    // ...
}
// onko instantioinnin paikka tässä?

void another() {
    int size = sizeof(asi) / sizeof(SmallInt);
    min( &asi[0], size );
}
// Vai täällä?
```

Sattumalta on olemassa instantiointipaikka jokaisen instantiointia käyttävän funktiomäärittelyn jälkeen. Kääntäjä on vapaa valitsemaan minkä tahansa näistä instantiointipaikoista funktiomallin instantioimiseksi. Tämä kertoo, että pitää olla varovainen, kun järjestämme koodiamme ja sijoittelemme nimiresoluutiota varten kaikki tarvittavat esittelyt, jotka riippuvat malliparamet-

rista ennen mallin ensimmäistä instantiointipaikkaa. On siksi hyvä ajatus laittaa kaikki tarvittavat esittelyt käytettäviksi otsikkotiedostoon ja ottaa se mukaan ennen kuin yhtäkään mallin instantiointia on käytetty. Esimerkiksi:

```
#include <primer.h>
// user.h sisältää instantioinnin tarvitsemat esittelyt
#include "user.h"
void another();

SmallInt asi[4];

int main() {
    // ...
}
// min( SmallInt*, int ):in ensimmäinen instantiointipaikka

void another() {
    // ...
}
// min( SmallInt*, int ):in toinen instantiointipaikka
```

Mitä jos mallin instantiointia on käytetty useammassa kuin yhdessä tiedostossa? Mitä jos esimerkiksi funktio another() on eri tiedostossa kuin main()-funktio? Jokaisessa tiedostossa, jossa mallin instantiointia on käytetty, on olemassa instantiointipaikka. Kääntäjä on vapaa valitsemaan minkä tahansa noista tiedostoista funktiomallin instantioimiseen. Pitää olla myös silloin huolellinen, kun järjestämme koodiamme ja sijoitamme "user.h"-otsikkotiedoston jokaiseen tiedostoon, jossa funktiomallin instantiointia on käytetty. Tämä varmistaa, että min(SmallInt*,int):in instantiointi viittaa print(const SmallInt &)-funktioomme, kuten odotimme, huolimatta siitä, minkä instantiointipaikan kääntäjä todellisuudessa valitsee.

Harjoitus 10.13

Luettele kaksi nimiresoluution vaihetta mallimäärittelyssä. Selitä, kuinka ensimmäinen vaihe koskee kirjastojen suunnittelijoita ja kuinka toinen vaihe tuo tarvittavaa joustavuutta mallin käyttäjille.

Harjoitus 10.14

Mihin esittelyihin nimet display ja SIZE viittaavat max(LongDouble*,SIZE):in instantioinnissa?

```
// ---- exercise.h ----
void display( const void* );
typedef unsigned int SIZE;

template<typename Type>
    Type max( Type* array, SIZE size )
```

```
{
    Type max_val = array[0];
    for ( SIZE i = 1; i < size; ++i )
        if ( array[i] > max_val )
            max_val = array[i];

    display( "Maximum value found: " );
    display( max_val );

    return max_val;
}
// ---- user.h ----
class LongDouble { /* ... */ };
void display( const LongDouble & );
void display( const char * );
typedef int SIZE;

// ---- user.C ----
#include <exercise.h>
#include "user.h"

LongDouble ad[7];

int main() {
    // aseta ad:in elementit

    // max( LongDouble*, SIZE ):in instantiointi
    SIZE size = sizeof(ad) / sizeof(LongDouble);
    max( &ad[0], size );
}
```

10.10 Nimiavaruudet ja funktiomallit

Kuten mikä tahansa globaalin viittausalueen määrittely, myös funktiomallin määrittely voidaan sijoittaa nimiavaruuteen. (Katso kohdista 8.5 ja 8.6 nimiavaruuksien käsittely.) Sellaisen mallimäärittelyn merkitys on sama, kuin jos se olisi 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:

```
// ---- primer.h ----
namespace cplusplus_primer {
    // mallin määrittely on piilotettu nimiavaruuteen
    template<class Type>
        Type min( Type* array, int size ) { /* ... */ }
}

// ---- user.C ----
```

```
#include <primer.h>
int ai[4] = { 12, 8, 73, 45 };

int main() {
    int size = sizeof(ai) / sizeof(ai[0]);

    // virhe: funktiota min() ei löydy
    min( &ai[0], size );

    using cplusplus_primer::min; // using-esittely
    // ok: viittaa funktioon min() nimiavaruudessa cplusplus_primer
    min( &ai[0], size );
}
```

Mitä tapahtuu, jos ohjelmamme käyttää mallia, joka on määritelty nimiavaruuteen, ja haluamme tehdä sille erikoistamisen? (Mallin eksplisiittiset erikoistamiset on esitelty kohdassa 10.6.) Haluamme esimerkiksi käyttää cplusplus_primer-nimiavaruuteen määriteltyä mallia min() niin, että se etsii pienimmän arvon taulukosta, joka muodostuu SmallInt-tyyppisistä olioista. Huomaamme kuitenkin, että cplusplus_primer-nimiavaruuden mallimäärittely ei aivan toimi. Mallimäärittelyn vertailu näyttää tältä:

```
if ( array[i] < min_val )
```

Tämä lause vertailee kahta SmallInt-tyyppistä luokkaoliota käyttäen pienempi kuin (<) -operaattoria. Tätä operaattoria ei voi käyttää kahteen luokkaolioon, ellei SmallInt-luokkaan ole määritelty ylikuormitettua operator<()-operaattoria. (Tulemme näkemään luvussa 15, kuinka ylikuormitettuja operaattoreita määritellään.) Olettakaamme, että haluamme määritellä erikoistamisen min()-funktiomallille niin, että se käyttää funktiota nimeltään compareLess() löytääkseen minimiarvon SmallInt-olioiden taulukosta. Seuraavassa on compareLess()-funktiomme esittely:

```
// vertailufunktio SmallInt-olioille
// palauttaa arvon true, jos parm1 on pienempi kuin parm2
bool compareLess( const SmallInt &parm1, const SmallInt &parm2 );
```

Miltä tämän funktion määrittely näyttäisi? Jotta tähän voitaisiin vastata, pitää katsoa SmallInt-luokkamme määrittelyä hieman tarkemmin. Voimme määritellä SmallInt-luokkaamme olioita, jotka voivat sisältää samoja arvoja kuin 8-bittinen unsigned char; tarkoittaa arvoja väliltä 0 – 255. Sen lisätoiminto on, että se “saa kiinni” ali- ja ylivuotovirheet. Sen lisäksi haluamme sen käyttäytyvän samaan tapaan kuin unsigned char. SmallInt-luokan määrittely näyttää tältä:

```
class SmallInt {
public:
    SmallInt( int ival ) : value( ival ) { }
    friend bool compareLess( const SmallInt &, const SmallInt & );
private:
    int value; // data member
};
```

Tässä luokan määrittelyssä on muutama asia, joita tulisi käsitellä. Ensiksi, luokassa on yksi yksityinen value-tietojäsen. Tämä on se, johon tallennetaan SmallInt-tyyppisen olion arvo. Luokka sisältää myös muodostajan:

```
// SmallInt-luokan muodostaja
SmallInt( int ival ) : value( ival ) { }
```

Tällä muodostajalla on yksi parametri, ival. Ainoa toimenpide, jonka se suorittaa, on luokan value-tietojäsenen alustaminen ival-parametrinsa avulla.

Nyt voimme vastata aikaisempaan kysymykseen. Kuinka compareLess()-funktiomme on määritelty? Funktio vertailee kahden SmallInt-parametrinsa value-tietojäseniä kuten seuraavassa:

```
// palauttaa arvon true, jos parm1 on pienempi kuin parm2
bool compareLess( const SmallInt &parm1, const SmallInt &parm2 ) {
    return parm1.value < parm2.value;
}
```

Huomaa kuitenkin, että value on SmallInt-luokan yksityinen tietojäsen. Kuinka tämä globaali funktio voi viitata yksityiseen tietojäseneen rikkomatta SmallInt-luokan kapselointia ja saamatta aikaan käännöksenaikaista virhettä? Jos katsot SmallInt-luokan määrittelyä, huomaat, että luokan määrittely on esitellyt globaalin compareLess()-funktion ystävänä (*friend*). Kun funktio on luokan ystävä, se voi viitata luokan yksityisiin jäseniin kuten compareLess()-funktiomme tekee. (Luokkien ystäviä katsotaan tarkemmin kohdassa 15.2.)

Olemme nyt valmiit määrittelemään min()-mallin erikoistamisen. Se käyttää compareLess()-funktiota seuraavasti:

```
// min():in erikoistaminen SmallInt-olioiden taulukoille
template<> SmallInt min<SmallInt>( SmallInt* array, int size )
{
    SmallInt min_val = array[0];
    for ( int i = 1; i < size; ++i )
        // vertailussa käytetään compareLess()-funktiotamme
        if ( compareLess( array[i], min_val ) )
            min_val = array[i];

    print( "Minimum value found: " );
    print( min_val );

    return min_val;
}
```

Missä erikoistaminen tulisi esitellä? Mitä sanot tästä?:

```
// ---- primer.h ----
namespace cplusplus_primer {
    // mallin määrittely piilotettu nimiavaruuteen
    template<class Type>
        Type min( Type* array, int size ) { /* ... */ }
```

```
}
// ---- user.h ----
class SmallInt { /* ... */ };
void print( const SmallInt & );
bool compareLess( const SmallInt &, const SmallInt & );

// ---- user.C ----
#include <primer.h>
#include "user.h"

// virhe: ei ole erikoistaminen: cplusplus_primer::min()
template<> SmallInt min<SmallInt>( SmallInt* array, int size )
    { /* ... */ }
// ...
```

Valitettavasti tämä koodi ei aivan tee sitä. Funktiomallin eksplisiittisen erikoistamisen esittely pitää olla nimiavaruudessa, jossa geneerinen malli on määritelty. Sitten min():in erikoistaminen pitää määritellä cplusplus_primer-nimiavaruuteen. On olemassa kaksi tapaa päästä tähän ohjelmassamme.

Muista, että nimiavaruuden määrittelyt voivat olla epäyhtenäisiä. Voimme avata cplusplus_primer-nimiavaruuden määrittelyn uudelleen ja lisätä erikoistamisen määrittelyn kuten seuraavassa:

```
// ---- user.C ----
#include <primer.h>
#include "user.h"

namespace cplusplus_primer {
    // cplusplus_primer::min():in erikoistaminen
    template<> SmallInt min<SmallInt>( SmallInt* array, int size )
        { /* ... */ }
}
SmallInt asi[4];

int main() {
    // aseta asi:n elementit käyttäen set()-jäsenfunktiota

    using cplusplus_primer::min; // using-esittely
    int size = sizeof(asi) / sizeof(SmallInt);
    // min(SmallInt*,int):in instantiointi
    min( &asi[0], size );
}
```

Tai voimme määritellä erikoistamisen samalla tavalla kuin määrittelemme minkä tahansa nimiavaruuden jäsenen nimiavaruuden määrittelyn ulkopuolelle: tarkentamalla nimiavaruuden jäsenen nimi sitä ympäröivän nimiavaruuden nimellä.

```
// ---- user.C ----
#include <primer.h>
#include "user.h"

// cplusplus_primer::min():in erikoistaminen
// erikoistamisen nimi on tarkennettu
template<> SmallInt cplusplus_primer::
            min<SmallInt>( SmallInt* array, int size )
    { /* ... */ }
// ...
```

Tästä syystä, koska olemme mallimäärittelyjä sisältävän kirjaston käyttäjiä, jos haluamme laittaa malleille erikoistamisia kirjastoon, pitää varmistua siitä, että niiden määrittelyt on sijoitettu oikein nimiavaruuksiin, jotka sisältävät alkuperäiset mallimäärittelyt.

Harjoitus 10.15

Sijoitamme nyt harjoituksen 10.14 <exercise.h>-otsikkotiedoston sisällön cplusplus_primer-nimiavaruuteen. Kuinka muuttaisit main()-funktiota niin, että se voisi instantioida funktiomallin max(), joka sijaitsee cplusplus_primer-nimiavaruudessa?

Harjoitus 10.16

Jos jälleen viittaamme harjoitukseen 10.14 ja oletamme, että <exercise.h>-otsikkotiedoston sisältö on sijoitettu cplusplus_primer-nimiavaruuteen, haluamme erikoistaa funktiomallin max() taulukoille, jotka muodostuvat LongDouble-luokan olioista. Haluamme, että mallin erikoistaminen käyttää määriteltyä compareGreater()-funktiota kahden LongDouble-luokkatyypin olion vertailuun kuten tässä:

```
// vertailufunktio LongDouble-olioille
// palauttaa arvon true, jos parm1 on suurempi kuin parm2
bool compareGreater( const LongDouble &parm1,
            const LongDouble &parm2 );
```

LongDouble-luokkamme määrittely näyttää tältä:

```
class LongDouble {
public:
    LongDouble(double dval) : value(dval) { }
    void set(double dval) { value = dval; }
    friend bool compareGreater( const LongDouble &,
                    const LongDouble & );
private:
    double value;
};
```

Tee määrittely compareGreater()-funktiolle ja max():in erikoistamiselle, joka käyttää tätä funktiota. Kirjoita main()-funktio, joka asettaa ad-taulukon elementit ja kutsuu sitten max():in erikoistamista saadakseen maksimiarvon ad-taulukosta; arvot, joilla ad-taulukon elementit alustetaan, tulisi saada lukemalla cin-vakiosyöttöä.

10.11 Esimerkki funktiomallista

Tässä kohdassa on esimerkki siitä, kuinka funktiomalleja voitaisiin määritellä ja käyttää. Esimerkissä määritellään funktiomalli sort(), jota käytetään taulukon elementtien lajitteluun. Itse taulukkoa edustaa Array-luokkamalli, joka esiteltiin kohdassa 2.5. Funktiomallia sort() voidaan siten käyttää minkä tahansa tyyppisen taulukon elementtien lajitteluun.

Näimme luvussa 6, että C++-vakiokirjastossa on säilötyyppi nimeltään vektori (*vector*), joka käyttäytyy melko samalla tavalla kuin kohdassa 2.5 määritelty Array-tyyppi. Luvussa 12 esitellään geneerisiä algoritmeja, joita voidaan käyttää luvussa 6 kuvattujen säilötyyppien käsittelyyn. Eräs näistä algoritmeista on nimeltään sort() ja sitä voidaan käyttää vektorin sisällön lajitteluun. Tässä kohdassa määrittelemme oman “geneerisen sort()-algoritmin” Array-luokkamme käsittelyä varten. Tässä näkemämme versio on paljon yksinkertaisempi kuin se, joka on C++-vakiokirjastossa.

sort()-funktiomallimme Array-luokkaa varten on määritelty seuraavasti:

```
#include "Array.h"

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

        for (;;) {
            while ( min( array[++lo], elem ) != elem && lo < high ) ;
            while ( min( array[--hi], elem ) == elem && hi > low ) ;

            if (lo < hi)
                swap( array, lo, hi );
            else break;
        }

        swap( array, low, hi );
        sort( array, low, hi-1 );
        sort( array, hi+1, high );
    }
}
```

sort()-funktio käyttää kahta apufunktiota: min() ja swap(). Nämä molemmat funktiot pitää määritellä funktiomalleiksi, jotta niillä voitaisiin käsitellä kaikki argumenttityypit, joilla sort() tullaan instantioimaan. min() on määritelty funktiomalliksi niin, että voimme löytää taulukon kahden minkä tahansa tyyppisen elementin minimiarvon:

```
template <class Type>
    Type min( Type a, Type b ) {
        return a < b ? a : b;
}
```

swap() on määritelty funktiomalliksi niin, että voimme vaihtaa keskenään taulukon kaksi minkä tahansa tyyppistä elementtiä:

```
#include "Array.h"

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

Jotta voisimme varmistua, että sort()-funktiomallimme todella toimii, pitää taulukon sisältö näyttää lajittelun jälkeen. Koska display()-funktion pitää pystyä käsittelemään mikä tahansa Array-luokkamallista instantioitu taulukko, myös se pitää määritellä funktiomalliksi:

```
#include <iostream>

template <class elemType>
    void display( Array<elemType> &array )
{ // tulostusmuoto: < 0 1 2 3 4 5 >

    cout << "< ";
    for ( int ix = 0; ix < array.size(); ++ix )
        cout << array[ix] << " ";
    cout << ">\n";
}
```

Tässä esimerkissä käytämme mukaan ottavaa käännösmuotoa ja sijoitamme kaikki funktiomallimme Array.h-otsikkotiedostoon Array-luokkamallimme määrittelyn perään.

Seuraava vaihe on kirjoittaa funktio, jolla kokeillaan näitä funktiomalleja. sort()-funktiolle välitetään vuorollaan double-, int- ja merkkijonotaulukko. Ohjelma on tässä:

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

double da[10] = {
    26.7, 5.7, 37.7, 1.7, 61.7, 11.7, 59.7,
    15.7, 48.7, 19.7 };

int ia[16] = {
    503, 87, 512, 61, 908, 170, 897, 275, 653,
    426, 154, 509, 612, 677, 765, 703 };

string sa[11] = {
    "a", "heavy", "snow", "was", "falling", "when",
    "they", "left", "the", "police", "station" };

int main() {

    // kutsu muodostajaa alustamaan arrd
    Array<double> arrd( da, sizeof(da)/sizeof(da[0]) );

    // kutsu muodostajaa alustamaan arri
    Array<int> arri( ia, sizeof(ia)/sizeof(ia[0]) );

    // kutsu muodostajaa alustamaan arrs
    Array<string> arrs( sa, sizeof(sa)/sizeof(sa[0]) );

    cout << "sort array of doubles (size == "
        << arrd.size() << ")" << endl;
    sort(arrd);
    display(arrd);

    cout << "\sort array of ints (size == "
        << arri.size() << ")" << endl;
    sort(arri);
    display(arri);

    cout << "\sort array of strings (size == "
        << arrs.size() << ")" << endl;
    sort(arrs);
    display(arrs);

    return 0;
}
```

Kun ohjelma käännetään ja suoritetaan, se generoi seuraavan tulostuksen (taulukon tulostus on katkaistu käsin, jotta se sopisi kirjan sivulle):

```
sort array of doubles (size == 10)
< 1.7 5.7 11.7 14.9 15.7 19.7 26.7
  37.7 48.7 59.7 61.7 >

sort array of ints (size == 16)
< 61 87 154 170 275 426 503 509 512
  612 653 677 703 765 897 908 >

sort array of strings (size == 11)
< "a" "falling" "heavy" "left" "police" "snow"
  "station" "the" "they" "was" "when" >
```

C++-vakiokirjastossa määriteltyjen geneeristen algoritmien joukosta (ja myös luvusta 12) löydät myös `min()`- ja `swap()`-funktiot. Luvussa 12 otamme selvää, kuinka niitä voidaan käyttää ohjelmissamme.