9

Ylikuormitetut funktiot

Nyt, kun tiedämme, kuinka funktioita esitellään ja määritellään ja kuinka käytämme funktioita ohjelmissamme, katsomme tässä luvussa eräitä erikoislaatuisia funktioita, joita C++ tukee: ylikuormitetut funktiot. Kaksi funktiota ovat ylikuormitettuja, jos niillä on sama nimi, ne ovat esiteltyinä samalla viittausalueella ja niillä on erilaiset parametriluettelot. Katsomme tässä luvussa ensin, kuinka joukko ylikuormitettuja funktioita esitellään ja miksi on hyödyllistä tehdä niin. Sitten katsomme lyhyesti, kuinka ylikuormitetun funktion ratkaisu etenee — kuinka funktion kutsu ratkaistaan yhteen ylikuormitettuun funktioon muiden joukosta. Funktion ylikuormituksen ratkaisu on eräs C++:n monimutkaisimmista aiheista. Niille, jotka haluavat ymmärtää sen yksityiskohtaisemmin, on tämän luvun lopussa kaksi lisäkohtaa, joissa kuvataan argumenttien tyyppikonversiot ja funktion ylikuormituksen ratkaisu täydellisemmin.

9.1 Ylikuormitettujen funktioiden esittelyt

Nyt, kun tiedämme, kuinka funktioita esitellään ja määritellään ohjelmiimme, katsomme uutta piirrettä funktioista, joita C++ tukee: *ylikuormitetut* (*overloaded*) funktiot. *Funktion ylikuormitus* mahdollistaa, että useammalla funktiolla, jotka tekevät yhteisen operaation eri parametrityypeillä, on sama nimi.

Jos olet kirjoittanut matemaattisia lausekkeita ohjelmointikielellä, olet käyttänyt esimääriteltyä ylikuormitettua funktiota. Esimerkiksi lauseke

```
1 + 3
```

käynnistää yhteenlaskuoperaation kahdelle kokonaislukuoperandille, kun taas lauseke

```
1.0 + 3.0
```

käynnistää eri yhteenlaskuoperaation, joka käsittelee liukulukuoperandeja. Operaatio, joka todellisuudessa tehdään, on läpinäkyvä käyttäjälle. Yhteenlaskuoperaatio on ylikuormitettu käsittelemään erityyppisiä operandeja. On kääntäjän eikä ohjelmoijan vastuulla erottaa eri operandit ja käyttää niihin sopivaa operaatiota tyyppien perusteella.

Tässä luvussa näemme, kuinka määrittelemme omia ylikuormitettuja funktioita.

9.1.1 Miksi ylikuormittaa funktion nimeä?

Aivan kuten sisäinen yhteenlaskuoperaatio toimii, voimme määritellä funktiojoukon, joka tekee saman yleisen toimenpiteen, mutta jossa käytetään eri parametrityyppejä. Oletetaan esimerkiksi, että haluamme määritellä funktiot, jotka palauttavat suurimman parametriensa arvoista.

Ilman funktion nimen ylikuormitusmahdollisuutta pitää jokaiselle funktiolle antaa oma yksilöllinen nimi. Voisimme esimerkiksi määritellä joukon max()-funktioita kuten seuraavassa:

```
int i_max( int, int );
int vi_max( const vector<int> & );
int matrix_max( const matrix & );
```

Jokainen funktio tekee kuitenkin saman yleisen toimenpiteen; jokainen palauttaa suurimman arvon parametreistaan. Käyttäjän kannalta on vain yksi operaatio, joka päättelee suurimman arvon. Toteutuksen yksityiskohdat, eli kuinka se tehdään, ei kiinnosta kovinkaan paljon funktion käyttäjiä.

Tämä sanallinen monimutkaisuus ei ole sen ongelman ydin, jossa päätellään joukon suurinta arvoa, vaan se pikemminkin heijastelee ohjelmointiympäristön rajoitusta, jossa samalla viittausalueella esiintyvien nimien pitää viitata yksilölliseen kohteeseen (yksilöllinen olio, funktio, luokan tyyppi jne.). Sellainen sanoihin liittyvä monimutkaisuus on käytännön ongelma ohjelmoijalle, jonka täytyy muistaa tai etsiä jokainen nimi. Funktion ylikuormitus vapauttaa ohjelmoijan tästä sanojen monimutkaisuudesta.

Kun funktiota ylikuormitetaan, ohjelmoija voi kirjoittaa yksinkertaisesti seuraavaa:

```
int ix = max( j, k );

vector<int> vec;
// ...
int iy = max( vec );
```

Tämä tekniikka osoittaa suurimman arvonsa monissa yhteyksissä.

9.1.2 Kuinka funktion nimeä ylikuormitetaan?

C++:ssa kahdelle tai useammalle funktiolle voidaan antaa sama nimi edellyttäen, että jokainen parametriluettelo on yksilöllinen joko lukumäärältään tai parametrien tyypeiltään. Seuraavat esittelyt ovat ylikuormitetun max()-funktion esittelyitä:

```
int max( int, int );
int max( const vector<int> & );
int max( const matrix & );
```

max()-funktiolle tarvitaan erilliset määrittelyt jokaiselle ylikuormitusesittelylle, jolla on yksilöllinen parametriluettelo.

Kun funktion nimi on esitelty useammin kuin kerran tietyllä viittausalueella, kääntäjä tulkitsee toisen (ja sitä seuraavat) esittelyn seuraavasti:

- Jos kahden funktion parametriluettelot eroavat toisistaan joko parametriensa lukumäärän tai tyyppien perusteella, pidetään noita kahta funktiota ylikuormitettuina. Esimerkiksi:

```
// ylikuormitetut funktiot
void print( const string & );
void print( vector<int> & );
```

- Jos kahden funktion esittelyiden paluutyypit ja parametriluettelot ovat täsmälleen samanlaisia, toista esittelyä pidetään ensimmäisen uudelleenesittelynä. Esimerkiksi:

```
// esittelee saman funktion
void print( const string &str );
void print( const string & );
```

  Parametrien nimillä ei ole merkitystä, kun parametriluetteloita verrataan.

- Jos kahden funktion parametriluettelot ovat täsmälleen samanlaisia, mutta niiden paluutyypit ovat erilaisia, pidetään toista esittelyä ensimmäisen funktion virheellisenä esittelynä ja se saa aikaan käännöksenaikaisen virheen. Esimerkiksi:

```
unsigned int max( int i1, int i2 );
int max( int , int ); // virhe: vain paluutyyppi on erilainen
```

  Funktion palyytyyppi ei riitä erottamaan kahta ylikuormitettua funktiota.

- Jos kahden funktion parametriluettelot eroavat toisistaan vain oletusargumenttiensa perusteella, toista esittelyä pidetään ensimmäisen uudelleenesittelynä. Esimerkiksi:

```
// esittelee saman funktion
int max( int *ia, int sz );
int max( int *, int = 10 );
```

Typedef-nimellä saadaan vaihtoehtoinen nimi olemassa olevalle tyypille; se ei luo uuttaa tietotyyppiä. Siitä syystä kahden funktion parametriluettelot, jotka eroavat toisistaan vain siinä, että toinen käyttää typedef-nimeä ja toinen tyyppiä, jota typedef vastaa, eivät ole erilaisia. Seuraavia kahta calc()-funktion esittelyä kohdellaan aivan kuin niillä olisi täsmälleen samanlaiset parametriluettelot. Vaikka toisessa esittelyssä on samanlainen parametriluettelo, se johtaa käännöksenaikaiseen virheeseen, koska se esittelee erilaisen paluutyypin ensimmäisestä esittelystä.

```
// typedef ei esittele uutta tyyppiä
typedef double DOLLAR;

// virhe: sama parametriluettelo, ei paluutyypit
extern DOLLAR calc( DOLLAR );
extern int calc( double );
```

Kun parametrin tyyppi on const tai volatile, ei määrettä const tai volatile oteta huomioon, kun eri funktioiden esittelyitä tunnistetaan. Esimerkiksi seuraavat kaksi esittelyä esittelevät saman funktion:

```
// esittelee saman funktion
void f( int );
void f( const int );
```

Tosiasialla, että parametri on const, on merkitystä vain funktion määrittelyssä: se tarkoittaa, että funktion runko ei voi muuttaa parametrin arvoa. Kuitenkin, kun argumentti välitetään arvona, tämä on täysin läpinäkyvää funktion käyttäjälle: käyttäjä ei koskaan näe muutoksia, joita funktio tekee arvona välitetylle argumentille. (Argumenttien välitystä arvona ja muita argumenttien välitysmetodeja käsitellään kohdassa 7.3.) Kun parametri esitellään const-tyyppiseksi ja välitetään arvona, se ei mitenkään muuta funktiolle välitettäviä argumentteja. Mitä tahansa int-tyyppiä voidaan käyttää f(int)-funktion kutsumiseen kuten myös f(const int)-funktion. Koska molemmat funktiot hyväksyvät samanlaisen argumentin, juuri nähdyt esittelyt eivät tee niistä ylikuormitettuja funktioita. Funktio f() voidaan määritellä näin

```
void f( int i ) { }
```

tai näin

```
void f( const int i ) { }
```

Näiden molempien määrittelyiden laittaminen samaan ohjelmaan on kuitenkin virhe, koska ne määrittelevät saman funktion kahdesti.

Kuitenkin, jos käytetään const- tai volatile-määrettä tyyppiin, johon viittaa osoitin- tai viittausparametri, silloin const- tai volatile-määre otetaan huomioon eri funktioiden esittelyitä tunnistettaessa.

```
// esittelee eri funktiot
void f( int* );
void f( const int* );

// esittelee myös eri funktiot
void f( int& );
void f( const int& );
```

9.1.3 Milloin funktion nimeä ei kannata ylikuormittaa?

Milloin ei hyödytä ylikuormittaa funktion nimeä? Se ei ole hyödyllistä silloin, kun eri funktion nimet tuottaisivat tietoa, joka saisi ohjelman helpommin ymmärrettäväksi. Tässä on joitakin esimerkkejä. Seuraava funktiojoukko käyttää yleistä tietoabstraktiota. Ne voivat aluksi näyttää todennäköisiltä ehdokkailta ylikuormitukseen:

```
void setDate( Date&, int, int, int );
Date &convertDate( const string & );
void printDate( const Date& );
```

Nämä funktiot käyttävät samaa tietotyyppiä — nimittäin Date-luokkaa — mutta eivät tee samaa operaatiota. Tässä tapauksessa sekavuus, joka liittyy funktioiden nimiin, on aiheutunut ohjelmoijan tottumuksesta käyttää funktioiden nimeämiseen operaatioita ja tietotyyppiä.

C++:n luokkamekanismi tekee tämänkaltaisesta käytännöstä tarpeettoman. Sen sijaan näistä funktioista tulisi tehdä Date-luokan jäsenfunktioita, ja koska jokainen jäsenfunktio tekee eri operaation, tulisi sen nimen edustaa tuota operaatiota. Esimerkiksi:

```
#include <string>
class Date {
public:
    set( int, int, int );
    Date &convert( const string & );
    void print();

    // ...
};
```

Tässä on toinen esimerkki. Seuraavan Screen-luokan viiden jäsenfunktion joukko tekee useita eri siirto-operaatioita Screen-luokan kohdistimella. Voi aluksi näyttää siltä, että on parempi ylikuormittaa tämä funktiojoukko move()-nimen alle:

```
Screen& moveHome();
Screen& moveAbs( int, int );
Screen& moveRel( int, int, char *direction );
Screen& moveX( int );
Screen& moveY( int );
```

Kahta viimeistä ilmentymää ei voi ylikuormittaa, koska niiden parametriluettelot ovat täsmälleen samat. Jotta saisimme yksilöllisen merkityksen, tiivistämme nuo kaksi funktiota seuraavasti:

```
// yhdistetyt funktiot moveX() ja moveY()
Screen& move( int, char xy );
```

Jokaisella funktiolla on nyt yksilöllinen parametriluettelo, joka mahdollistaa niiden ylikuormittamisen nimellä move(). Kuitenkin kriteeriemme mukaan näiden funktioiden ylikuormittaminen on huono ajatus: eri funktionimet antavat tietoa, joka muutoin häviäisi ja saisi ohjelman epäselvemmäksi. Vaikka kohdistimen liikuttelu on näiden funktioiden yhteinen operaatio, jokainen tietty siirto on yksilöllinen näiden funktioiden joukossa. Esimerkiksi moveHome() edustaa erityistilannetta kohdistimen liikuttelussa. Kumpi kahdesta kutsusta on ohjelman lukijan helpompi ymmärtää? Kumpi kahdesta kutsusta on luokan Screen käyttäjälle helpompi muistaa?

```
// kumpi on helpompi ymmärtää?
myScreen.home(); // mielestämme tämä!
myScreen.move();
```

Ylikuormittaminen voi olla joskus tarpeetonta, jolloin funktioiden eri määrittelyjä ei tarvita. Joissain tapauksissa oletusargumentit mahdollistavat, että useampi funktion esittely voidaan tiivistää yhteen funktioon. Esimerkiksi kaksi kohdistinfunktiota

```
moveAbs(int,int);
moveAbs(int,int,char*);
```

erotetaan toisistaan kolmannen parametrin, char*, läsnä- tai poissaolosta. Jos näiden kahden funktion toteutukset ovat lähes samanlaisia ja jos char*-parametrin oletusargumentista voidaan päätellä silloin, kun se välitetään funktiolle, että sillä on sama merkitys, kuin jos parametria ei olisi annettu, voidaan nämä kaksi funktiota yhdistää. Ja juuri sellainen oletusargumentti on olemassa tässä tapauksessa — osoitin, jonka arvo on 0:

```
move( int, int, char* = 0 );
```

Ohjelmoijille olisi parasta, etteivät he ajattelisi jokaisen kielen uuden piirteen olevan kuin uusi vuori, jonne kiivetä. Piirteen käytön tulisi johtua sovelluksen logiikasta eikä vain siitä, että se on olemassa. Ohjelmoijien ei tulisi pyrkiä väkisin laittamaan ylikuormitettuja funktioita ohjelmiinsa. Niitä tulisi toteuttaa vain siellä, missä niiden käyttö tuntuu luonnolliselta.

9.1.4 Viittausalueen ylikuormittaminen

Kaikki ylikuormitetun funktiojoukon jäsenet esitellään samalla viittausalueella. Esimerkiksi paikallisena esitelty funktio peittää globaalilla viittausalueella esitellyn funktion ylikuormituksen sijasta. Esimerkiksi:

```
#include <string>
void print( const string & );
void print( double ); // ylikuormittaa funktion print()

void fooBar( int ival )
{
    // eri viittausalue: peittää molemmat print()-funktion ilmentymät
    extern void print( int );

    // virhe: print( const string & ) peittyy tällä viittausalueella
    print( "Value: " );
    print( ival );  // ok: print( int ) on näkyvissä
}
```

Ylikuormitettujen funktioiden joukko voidaan esitellä myös luokassa. Koska jokainen luokka hoitaa oman viittausalueensa, kahden eri luokan funktiojäsenet eivät ylikuormita toisiaan. Luokan jäsenfunktiot kuvataan luvussa 13. Luokan jäsenfunktioiden ylikuormituksen resoluutio kuvataan luvussa 15.

Ylikuormitettujen funktioiden joukko voidaan esitellä myös nimiavaruudessa. Jokainen nimiavaruus myös hoitaa oman viittausalueensa, joten kahden eri nimiavaruuden funktiot eivät ylikuormita toisiaan. Esimerkiksi:

```
#include <string>
namespace IBM {
    extern void print( const string & );
    extern void print( double ); // ylikuormittaa funktion print()
}
namespace Disney {
    // eri viittausalue:
```

```
    // ei ylikuormita IBM:n print()-funktioita
    extern void print( int );
}
```

Using-esittelyt ja using-direktiivit ovat mekanismeja, joilla nimiavaruuden jäseniä voidaan saada näkyviin toisilla viittausalueilla. Näillä mekanismeilla on joitakin vaikutuksia ylikuormitettujen funktioiden esittelyihin. Using-esittelyt ja using-direktiivit on esitelty kohdassa 8.6.

Kuinka using-esittely toimii ylikuormitetulle funktiolle? Muista, että using-esittely esittelee aliasnimen nimiavaruuden jäsenelle viittausalueella, jossa using-esittely esiintyy. Mitä tapahtuu using-esittelyille seuraavassa ohjelmassa?

```
namespace libs_R_us {
    int max( int, int );
    int max( double, double );

    extern void print( int );
    extern void print( double );
}

// using-esittelyt:
using libs_R_us::max;
using libs_R_us::print( double ); // virhe

void func()
{
    max( 87, 65 );    // kutsuu funktiota libs_R_us::max( int, int )
    max( 35.5, 76.6 ); // kutsuu funktiota libs_R_us::max( double, double )
}
```

Ensimmäinen using-esittely esittelee molemmat libs_R_us::max()-funktiot globaalilla viittausalueella. Siten on mahdollista kutsua kumpaa tahansa max()-funktiota func()-funktiosta. Funktion kutsun argumenttien tyypit ratkaisevat kutsuttavan funktion. Toinen using-esittely on virheellinen. Käyttäjä ei voi määrittää parametriluetteloa funktion using-esittelyssä. Ainoa kelvollinen using-esittely libs_R_us::print()-funktiolle on

```
using libs_R_us::print;
```

Using-esittely esittelee aina aliakset *kaikille* ylikuormitetuille funktioille. Miksi tämä rajoitus on tarpeellinen? Tämä rajoitus takaa, että libs_R_us-nimiavaruuden rajapinta pysyy eheänä. On selvää, että funktion kuten

```
print( 88 );
```

nimiavaruuden kirjoittaja olettaa, että kutsutaan libs_R_us::print(int)-funktiota. Kirjaston kirjoittaja teki eri funktiot syystä. Se, että sallitaan käyttäjien lisätä valikoidusti viittausalueelle yksi funktio ylikuormitettujen funktioiden joukosta, mutta ei kaikkia funktioita, voi johtaa ohjelman yllättävään käyttäytymiseen.

Mitä tapahtuu, jos using-esittely esittelee funktion viittausalueella, jossa on jo samanniminen funktio? Muista, että using-esittely on itsekin esittely. Se on sama kuin using-esittelyllä esi-

tellyt funktiot olisi esitelty siellä, missä using-esittely esiintyy. Tästä syystä using-esittelyllä esitellyt funktiot ylikuormittavat muiden funktioiden esittelyitä, joilla on sama nimi ja jotka ovat samalla viittausalueella, jossa using-esittely esiintyy. Esimerkiksi:

```
#include <string>
namespace libs_R_us {
    extern void print( int );
    extern void print( double );
}

extern void print( const string & );

// libs_R_us::print( int ) ja libs_R_us::print( double )
// ylikuormittavat funktion print( const string & )
using libs_R_us::print;

void fooBar( int ival )
{
    print( "Value: " ); // kutsuu globaalia funktiota print( const string & )
    print( ival );     // kutsuu funktiota libs_R_us::print( int )
}
```

Using-esittely lisää kaksi esittelyä globaalille viittausalueelle: yhden print(int)-funktiolle ja yhden print(double)-funktiolle. Nämä esittelyt ovat nimiavaruuden libs_R_us funktioiden aliaksia. Esittelyt on lisätty ylikuormitettujen print()-funktioiden joukkoa varten, joista globaali print(const string &)-funktio on jo jäsen. Kun fooBar()-funktiosta löytyy funktion kutsu, otetaan kaikki print()-funktiot huomioon.

Jos using-esittely esittelee funktion viittausalueella, jolla on jo samanniminen funktio ja parametriluettelo, using-esittely on silloin virheellinen. Using-esittely ei voi esitellä print(int)-funktiota aliakseksi funktiolle, joka on libs_R_us-nimiavaruudessa, jos siellä on jo olemassa funktio nimeltään print(int) globaalilla viittausalueella. Esimerkiksi:

```
namespace libs_R_us {
    void print( int );
    void print( double );
}

void print( int );
using libs_R_us::print; // virhe: print(int)-funktion uudelleenesittely

void fooBar( int ival )
{
    print( ival );     // kumpi print? ::print vai libs_R_us::print?
}
```

Nyt, kun olemme tutkineet, kuinka using-esittelyt vaikuttavat ylikuormitettuihin funktioihin, katsokaamme, kuinka niihin vaikuttavat using-direktiivit. Using-direktiivi saa nimiavaruuden jäsenen näyttämään siltä, kuin se olisi esitelty nimiavaruuden ulkopuolella. Kun

nimiavaruuden rajat poistetaan, using-direktiivi lisää esittelyt viittausalueelle, jossa nimiavaruus on määritelty. Jos tällä viittausalueella on esitelty funktio, jolla on sama nimi kuin nimiavaruuden jäsenfunktiolla, nimiavaruuden jäsen lisätään ylikuormitettujen joukkoon. Esimerkiksi:

```
#include <string>
namespace libs_R_us {
    extern void print( int );
    extern void print( double );
}

extern void print( const string & );

// using-direktiivi:
// print(int), print(double) ja print(const string &)
// ovat osa samaa ylikuormitettua joukkoa
using namespace libs_R_us;

void fooBar( int ival )
{
    print( "Value: " ); // kutsuu globaalia funktiota print(const string &)
    print( ival );     // kutsuu funktiota libs_R_us::print(int)
}
```

Tämä pätee myös, jos using-direktiivejä on useita. Eri nimiavaruuksien samannimiset jäsenfunktiot lisätään samaan ylikuormitettujen funktioiden joukkoon. Esimerkiksi:

```
namespace IBM {
    int print(int);
}
namespace Disney {
    double print(double);
}

// using-direktiivit:
// muodosta funktioiden ylikuormitettu joukko eri nimiavaruuksista
using namespace IBM;
using namespace Disney;

long double print(long double);

int main() {
    print(1);   // kutsuu funktiota IBM::print(int)
    print(3.1); // kutsuu funktiota Disney::print(double)
    return 0;
}
```

Funktion print() ylikuormitettujen funktioiden joukko sisältää funktiot print(int), print(double) ja print(long double). Nämä ovat osa ylikuormitettujen funktioiden joukosta, jotka otetaan huomioon kutsuissa main()-funktioon, vaikka ne esiteltiin alun perin eri nimiavaruuksien viittausalueilla.

Ylikuormitettu funktiojoukko on siten esitelty samalla viittausalueella, vaikka nämä esittelyt voidaan esitellä using-esittelyillä ja using-direktiiveillä, jotka saavat nimiavaruuden jäsenet näkyviksi aivan kuin ne olisi esitelty eri viittausalueilla.

9.1.5 Direktiivi extern "C" ja ylikuormitetut funktiot

Kuten olemme nähneet kohdassa 7.7, voimme käyttää linkitysdirektiiviä extern "C" ja määrittää, että funktio on kirjoitettu C-ohjelmointikielellä C++-ohjelmassamme. Kuinka extern "C" -linkitysdirektiivi vaikuttaa ylikuormitettujen funktioiden esittelyihin? Voivatko jotkut ylikuormitetuista funktioista olla C++-funktioita ja toiset C-funktioita?

Linkitysdirektiivi voidaan määrittää vain yhdelle ylikuormitetuista funktioista. Esimerkiksi ohjelma, jossa on mukana seuraavat kaksi esittelyä, on virheellinen:

```
// virhe: kaksi extern "C" -funktiota ylikuormitetuissa funktioissa
extern "C" void print( const char* );
extern "C" void print( int );
```

Seuraava calc()-funktion ylikuormitus kuvaa tyypillistä linkitysdirektiivin käyttöä ylikuormituksessa:

```
class SmallInt { /* ... */ };
class BigNum { /* ... */ };

// C-funktiota voidaan kutsua sekä C- että C++-ohjelmista
// C++-funktiot käsittelevät parametreja, jotka ovat C++-luokkia
extern "C" double calc( double );
extern SmallInt calc( const SmallInt& );
extern BigNum calc( const BigNum& );
```

C-kielistä calc()-funktiota voidaan kutsua C-ohjelmista ja C++-ohjelmista. Lisäfunktiot ovat C++-kielisiä luokkaparametreineen, joita voidaan kutsua vain C++-ohjelmista. Esittelyiden järjestys ei ole merkitsevä.

Linkitysdirektiivi ei vaikuta funktioon, joka valitaan kutsun yhteydessä; kutsutun funktion valintaan vaikuttavat vain parametrien tyypit. Valittu funktio on se, joka parhaiten vastaa funktion argumenttien tyyppejä. Esimerkiksi:

```
SmallInt si = 8;
int main() {
    calc( 34 ); // kutsuu C-kielistä funktiota calc( double )
    calc( si ); // kutsuu C++-kielistä funktiota calc( const SmallInt & )
    // ...
    return 0;
}
```

9.1.6 Osoittimet ylikuormitettuihin funktioihin

On mahdollista esitellä osoitin viittaamaan tiettyyn ylikuormitettuun funktioon. Kuinka se tehdään? Esimerkiksi:

```
extern void ff( vector<double> );
extern void ff( unsigned int );

// mihin funktioon pf1 viittaa?
void ( *pf1 )( unsigned int ) = &ff;
```

Koska ff() on ylikuormitettu funktio, kääntäjä ei tiedä, kumman funktion se valitsisi katsomalla vain alustuslauseketta: &ff. Valitakseen funktion, joka alustaa osoittimen, kääntäjä etsii ylikuormitettujen joukosta funktiota, jolla on sama paluutyyppi ja samanlainen parametriluettelo kuin funktiolla, johon osoitin viittaa. Edellisessä tapauksessa valitaan ff(unsigned int).

Mitä, jos yksikään funktio ei vastaa osoittimen tyyppiä täsmälleen? Ellei yksikään funktio täsmää, johtaa alustus käännöksenaikaiseen virheeseen. Esimerkiksi:

```
extern void ff( vector<double> );
extern void ff( unsigned int );

// virhe: ei täsmää: kelvoton parametriluettelo
void ( *pf2 )( int ) = &ff;

// virhe: ei täsmää: kelvoton paluutyyppi
double ( *pf3 )( vector<double> ) = &ff;
```

Sijoitukset toimivat samalla tavalla. Jos ylikuormitetun funktion osoite sijoitetaan funktion osoittimeen, käytetään funktion osoitintyyppiä hyväksi valittaessa funktiota sijoitusoperaattorin oikealta puolelta. Ja jos kääntäjä ei löydä funktiota, joka vastaa osoittimen tyyppiä, sijoitus on virheellinen; tarkoittaa, että kahden funktio-osoittimen välillä ei ole olemassa tyyppikonversioita:

```
matrix calc( const matrix & );
int calc( int, int );

int ( *pc1 )( int, int ) = 0;
int ( *pc2 )( int, double ) = 0;

// ...
// ok: matches int calc( int, int );
pc1 = &calc;

// virhe: ei täsmää: toinen parametrityyppi ei kelpaa
pc2 = &calc;
```

9.1.7 Tyyppiturvallinen linkitys

Ylikuormitus sallii saman funktion nimen esiintymisen useasti eri parametriluetteloilla. Tämä on nimien käytön kannalta mukavuus, joka pätee ohjelman lähdekoodin tasolla. Useimpien käännösjärjestelmien taustakomponentit kuitenkin vaativat, että funktion pitää olla yksilöllisesti nimetty. Monet linkitysohjelmat ratkaisevat ulkoiset viittaukset nimien mukaan. Jos linkitysohjelma näkee kaksi tai useamman ilmentymän nimestä print, se ei voi analysoida tyyppejä erottaakseen olioita toisistaan (käännöksen tässä vaiheessa tyyppitieto on usein jo hävinnyt). Sen sijaan linkitysohjelma ilmoittaa virheestä, että print on määritelty useasti, ja keskeyttää.

Tämän ongelman käsittelemiseksi jokainen funktion nimi parametriluetteloineen *koodataan* yksilölliseksi sisäiseksi nimeksi. Käännösjärjestelmän taustakomponentit näkevät vain tämän koodatun nimen. Nimen muunnoksen täsmälliset yksityiskohdat eivät ole merkityksellisiä; ne vaihtelevat todennäköisesti toteutuksesta toiseen. Yleinen algoritmi koodaa parametrien tyypit ja lukumäärän ja lisää sen funktion nimeen.

Kuten olemme nähneet globaalien funktioiden kohdalla 8.2, tämä erityinen koodaus myös varmistaa, että linkitysohjelma ei pidä samana kahden funktion esittelyitä, joilla on sama nimi, mutta eri parametriluettelot, ja jotka sijaitsevat eri tiedostoissa. Koska tämä koodaus helpottaa linkitysvaihetta erottamaan ylikuormitetut funktiot ohjelmasta, sitä kutsutaan *tyyppiturvalliseksi linkitykseksi* (*type-safe linkage*).

Tätä erityistä koodausta ei käytetä funktioihin, jotka on esitelty linkitysdirektiivillä extern "C". Tämä on syy, miksi vain yksi funktio ylikuormitetusta funktiojoukosta voidaan esitellä extern "C": kaksi extern "C" -funktiota eri parametriluetteloineen näkyvät linkitysohjelmille samanlaisina funktioina.

Harjoitus 9.1

Miksi esittelisit ylikuormitettuja funktioita?

Harjoitus 9.2

Kuinka pitäisi esitellä ylikuormitettujen funktioiden joukko seuraavasta error()-funktiosta, joka käsittelisi seuraavat kutsut?

```
int index;
int upperBound;
char selectVal;
// ...
error( "Array out of bounds: ", index, upperBound );
error( "Division by zero" );
error( "Invalid selection", selectVal );
```

Harjoitus 9.3

Selitä toisen esittelyn vaikutus jokaisesta seuraavista esittelyistä.

```
(a) int calc( int, int );
    int calc( const int, const int );

(b) int get();
    double get();

(c) int *reset( int * );
    double *reset( double * );

(d) extern "C" int compute( int *, int );
    extern "C" double compute( double *, double );
```

Harjoitus 9.4

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

```
(a) void reset( int * );
    void (*pf)( void * ) = reset;

(b) int calc( int, int );
    int (*pf1)( int, int ) = calc;

(c) extern "C" int compute( int *, int );
    int (*pf3)( int*, int ) = compute;

(d) void (*pf4)( const matrix & ) = 0;
```

9.2 Ylikuormituksen ratkaisun kolme vaihetta

Funktion ylikuormituksen ratkaisu on prosessi, jossa funktion kutsu liitetään *yhteen* funktioon ylikuormitetuista funktioista. Se on prosessi, jossa useasta samannimisestä funktiosta valitaan yksi funktion kutsussa määritettyjen argumenttien perusteella. Mietitäänpä esimerkkiä:

```
T t1, t2;
void f( int, int );
void f( float, float );

int main() {
      f( t1,t2 );
      return 0;
}
```

Tässä funktion ylikuormituksen ratkaisussa päätellään annetulla tyypillä T, kutsuuko f(t1,t2) funktiota f(int,int) vai funktiota f(float,float)**,** onko kutsu virheellinen, koska kumpaakaan näistä funktioista ei voida kutsua argumenteilla t1 ja t2, vai onko kutsu *moniselitteinen*, koska kutsussa määritetyt argumentit käyvät molempiin funktioihin yhtä hyvin.

Funktion ylikuormituksen ratkaisu on eräs monimutkaisimmista aiheista C++-ohjelmointikielessä. C++-ohjelmoinnin vasta-alkajat voivat hämmentyä aluksi yrittäessään ymmärtää kaikkia sen yksityiskohtia. Tästä syystä tämä kohta esittää vain yleiskatsauksen, kuinka funktion ylikuormituksen ratkaisu etenee, ja antaa sinulle tuntuman siitä, mitä tapahtuu. Niille, jotka haluavat tietää enemmän, seuraavat kaksi kohtaa kuvaavat yksityiskohtaisemmin funktion ylikuormituksen ratkaisun prosessin.

Funktion ylikuormituksen ratkaisussa on kolme vaihetta. Käytämme seuraavaa esimerkkiä kuvataksemme näitä kolmea vaihetta:

```
void f();
void f( int );
void f( double, double = 3.4 );
void f( char*, char* );

void main() {
    f( 5.6 );
    return 0;
}
```

Funktion ylikuormituksen ratkaisun kolme vaihetta ovat seuraavat:

1. Yksilöi ylikuormitettujen funktioiden joukko, jotka huomioidaan kutsua varten ja yksilöi kutsussa olevan argumenttiluettelon ominaisuudet.
2. Valitse funktiot ylikuormitettujen funktioiden joukosta, joita voidaan kutsua kutsussa määritetyillä argumenteilla niiden lukumäärän ja tyypin perusteella.
3. Valitse funktio, joka parhaiten vastaa kutsua.

Katsomme jokaista vaihetta vuorollaan.

Ensimmäinen vaihe funktion ylikuormituksen ratkaisussa yksilöi kutsussa huomioon otettavat ylikuormitetut funktiot. Tätä funktiojoukkoa kutsutaan *funktioehdokkaiksi*. Funktioehdokas on funktio, jolla on sama nimi kuin funktiolla, jota kutsutaan, ja jonka esittely on näkyvissä kutsuhetkellä. Esimerkissämme on neljä funktioehdokasta: f(), f(int), f(double, double) ja f(char*, char*).

Funktion ylikuormituksen ratkaisun ensimmäinen vaihe myös yksilöi funktion kutsun argumenttiluettelon ominaisuudet, joka tarkoittaa niiden lukumäärää ja tyyppejä. Esimerkissämme argumenttiluettelo muodostuu yhdestä double-tyyppisestä argumentista.

Funktion ylikuormituksen ratkaisun toinen vaihe valitsee vaiheessa 1 valituista funktioehdokkaista funktiot, joita voidaan kutsua kutsussa määritellyillä argumenteilla. Siten valittuja funktioita kutsutaan *elinkelpoisiksi funktioiksi*. Elinkelpoisen funktion parametrien lukumäärä on sama kuin kutsun argumenttien lukumäärä tai sillä on enemmän parametreja kuin kutsun argumenttiluettelossa ja jokaisella lisäparametrilla on vastaava oletusargumentti. Jotta funktio voisi olla elinkelpoinen, pitää olla olemassa *konversioita*, joilla voidaan konvertoida jokainen argumenttityyppi funktion parametriluettelon vastaavaksi tyypiksi.

Esimerkissä on kaksi elinkelpoista funktiota, joita voidaan kutsua määritetyllä argumenttiluettelolla.

- f(int) on elinkelpoinen funktio, koska sillä on vain yksi parametri, ja on olemassa konversio, jolla voidaan konvertoida argumentin double-tyyppi parametrin int-tyypiksi.
- f(double, double) on elinkelpoinen funktio, koska oletusargumentti on annettu funktion toiselle parametrille ja sen ensimmäinen parametri on double-tyyppiä, joka vastaa täysin argumentin tyyppiä.

Jos funktion ylikuormituksen ratkaisun prosessi ei löydä funktiota, jota voidaan kutsua annetulla argumenttiluettelolla, on funktion kutsu virheellinen. Silloin ei ole olemassa funktiota, joka vastaa kutsua, ja sanomme, että kyseessä on *täsmäämätön* tilanne.

Funktion ylikuormituksen ratkaisun kolmas vaihe muodostuu funktion valitsemisesta, joka vastaa funktion kutsua parhaiten. Tätä funktiota kutsutaan *parhaiten elinkelpoiseksi funktioksi* (kutsutaan myös usein *parhaiten täsmääväksi funktioksi*). Jotta tämä funktio löydettäisiin, konversiot, joita käytetään argumenttien tyyppien konvertoimiseksi vastaaviksi funktion parametreiksi, laitetaan *paremmuusjärjestykseen*. Parhaiten elinkelpoinen funktio on se, johon pätee seuraava:

1. Argumentteihin käytetyt konversiot eivät ole *huonompia* kuin konversiot, jotka ovat tarpeellisia mille tahansa muulle elinkelpoiselle funktiolle.
2. Joidenkin argumenttien konversiot ovat *parempia*, kuin konversiot, jotka ovat tarpeellisia samoille argumenteille, kun kutsutaan muita elinkelpoisia funktioita.

Tyyppikonversioita ja niiden paremmuusjärjestystä tutkitaan tarkemmin kohdassa 9.3. Tutkimme tässä lyhyesti konversioiden paremmuusjärjestystä esimerkin kautta. Kun elinkelpoista f(int)-funktiota tutkitaan, konversio, jota käytetään double-tyypin konvertoimiseen int-tyypiksi, on vakiokonversio. Kun elinkelpoista f(double, double)-funktiota tutkitaan, double-tyyppinen argumentti vastaa täsmälleen vastaavaa parametria. Koska täsmällinen vastaavuus on parempi kuin vakiokonversio — koska konversion tekemättömyys on parempi kuin mikään konversio — parhaiten elinkelpoiseksi funktioksi kutsulle valitaan f(double, double).

Jos funktion ylikuormituksen ratkaisu ei löydä parhainta elinkelpoista funktiota, on funktion kutsu *moniselitteinen*: kutsu ei vastaa yhtäkään elinkelpoista funktiota muita paremmin.

Kuten mainitsimme aikaisemmin, lisää funktion ylikuormituksen ratkaisun vaiheista löytyy kohdasta 9.4. Funktion ylikuormituksen ratkaisua käytetään myös silloin, kun kutsutaan ylikuormitettua luokan jäsenfunktiota tai operaattorifunktiota. Kohdassa 15.10 käsitellään funktion ylikuormituksen ratkaisun sääntöjä luokan jäsenfunktioille. Kohdassa 15.11 käsitellään funktion ylikuormituksen ratkaisun sääntöjä ylikuormitetuille operaattoreille. Funktion ylikuormituksen ratkaisu pitää ottaa huomioon, kun funktio generoidaan funktiomallista. Kohdassa 10.8 käsitellään, kuinka funktiomallit vaikuttavat funktion ylikuormituksen ratkaisuun.

Harjoitus 9.5

Mitä tapahtuu viimeisessä (kolmannessa) funktion ylikuormituksen ratkaisuvaiheessa?

9.3 Argumenttien tyyppikonversiot

Funktion ylikuormituksen ratkaisun toisessa vaiheessa kääntäjä yksilöi ja laittaa paremmuusjärjestykseen konversiot, joita voidaan käyttää funktion kutsun jokaisen argumentin konvertoimiseen vastaavaksi parametrin tyypiksi jokaisessa elinkelpoisessa funktiossa. Tämän paremmuusjärjestyksen kolme mahdollista tulosta ovat:

1. *Täsmällinen vastaavuus*. Argumentti vastaa täsmälleen funktion parametrin tyyppiä. Esimerkiksi seuraavien ylikuormitettujen funktioiden kolme print()-funktiota vastaavat täsmälleen seuraavia kolmea kyseisen funktion kutsua:

```
void print( unsigned int );
void print( const char* );
void print( char );

unsigned int a;

print( 'a' );    // vastaa funktiota print( char );
print( "a" );    // vastaa funktiota print( const char* );
print( a );      // vastaa funktiota print( unsigned int );
```

2. Vastaa *tyyppikonversion* avulla. Argumentin tyyppi ei suoraan vastaa parametrin tyyppiä, mutta se voidaan konvertoida sellaiseksi tyypiksi:

```
void ff( char );

ff( 0 ); // argumentti konvertoitu tyypistä int tyypiksi char
```

3. *Ei vastaavuutta*. Argumenttia ei saada vastaamaan esiteltyjen funktioiden parametria, koska tyyppikonversiota ei ole olemassa argumentin ja vastaavan funktion parametrin välillä. Seuraavien kahden print()-funktion kutsun jokainen argumentti johtaa siihen, että vastaavuutta ei löydy:

```
// print()-funktion esittelyt kuten edellä
int *ip;
class SmallInt { /* ... */ };
SmallInt si;

print( ip ); // virhe: ei täsmää
print( si ); // virhe: ei täsmää
```

Jotta argumentti vastaisi täsmälleen, sen ei tarvitse olla täsmälleen parametrin tyyppinen. On olemassa joitakin pieniä konversioita, joita argumenttiin voidaan käyttää. Täsmällisen vastaavuuden kategoriaan kuuluvat seuraavat konversiot:

1. Lvalue rvalueksi -konversio
2. Taulukko osoittimeksi -konversio
3. Funktio osoittimeksi -konversio
4. Määrekonversiot

Tutkimme näitä konversioita yksityiskohtaisemmin myöhemmin.

Kategoria “vastaa tyyppikonversion avulla” on kaikkein monimutkaisin näistä kolmesta kategoriasta. Pitää ottaa huomioon lukuisia erityyppisiä konversioita . Mahdolliset konversiot voidaan ryhmittää kolmeen kategoriaan: *ylennykset*, *vakiokonversiot* ja *käyttäjän määrittelemät konversiot*. Ylennykset ja vakiokonversiot käsitellään myöhemmin tässä luvussa. Käyttäjän määrittelemät konversiot esitellään myöhemmin sen jälkeen, kun olemme käsitelleet luokkia yksityiskohtaisemmin. Käyttäjän määrittelemä konversio tehdään *konversiofunktiolla*: jäsenfunktio, joka mahdollistaa luokan määritellä oma joukko “vakiokonversioita”. Luvussa 15 katsotaan käyttäjän määrittelemiä konversiofunktioita luokan ja ylikuormitetun funktion ratkaisun kannalta.

Kun valitaan parhaiten elinkelpoista funktiota kutsulle, kääntäjä valitsee funktion, jonka argumenttien tyyppikonversiot ovat “parhaita”. Tyyppikonversiot laitetaan paremmuusjärjestykseen seuraavasti: täsmällinen vastaavuus on parempi kuin ylennys, ylennys on parempi kuin vakiokonversio ja vakiokonversio on parempi kuin käyttäjän määrittelemä konversio. Tutkimme tyyppikonversioiden paremmuusjärjestykseen laittamista tarkemmin kohdassa 9.4, mutta koska kuvaamme mahdollisia tyyppikonversioita tässä, jotkin tämän kohdan esimerkit osoittavat yksinkertaiset tapaukset, kuinka tätä paremmuusjärjestykseen laittamista käytetään parhaan elinkelpoisen funktion valitsemiseksi.

9.3.1 Täsmällisen vastaavuuden yksityiskohdat

Täsmällisen vastaavuuden yksinkertaisin tapaus on, kun argumentit vastaavat parametrien tyyppejä täsmälleen. Esimerkiksi seuraavien kahden ylikuormitetun max()-funktion kaksi kutsua vastaavat täsmälleen kunkin ylikuormitetun funktion parametreja:

```
int max( int, int );
double max( double, double );

int i1;

void calc( double d1 ) {
    max( 56, i1 );   // vastaa täsmälleen funktiota max( int, int );
    max( d1, 66.9 ); // vastaa täsmälleen funktiota max( double, double );
}
```

Lueteltu tyyppi määrittelee yksilöllisen tyypin, joka vastaa täsmälleen vain luetellun joukon lueteltuja tyyppejä ja olioita, jotka on esitelty luetelluiksi tyypeiksi. Esimerkiksi:

```
enum Tokens { INLINE = 128; VIRTUAL = 129; };
Tokens curTok = INLINE;

enum Stat { Fail, Pass };

extern void ff( Tokens );
extern void ff( Stat );
extern void ff( int );

int main() {
    ff( Pass );   // vastaa täsmälleen funktiota ff( Stat )
    ff( 0 );      // vastaa täsmälleen funktiota ff( int )
    ff( curTok ); // vastaa täsmälleen funktiota ff( Tokens )
    // ...
}
```

Kuten aikaisemmin mainittiin, argumentti voi vastata täsmälleen, vaikka joitakin pienempiä tyyppikonversioita pitää tehdä, jotta argumentti saataisiin vastaavan funktion parametrin tyyppiseksi.

Ensimmäinen näistä on *lvalue rvalueksi* -konversio. Lvalue edustaa oliota, jota ohjelmassamme voidaan osoittaa ja josta arvo voidaan saada. Ellei oliota ole esitelty const-tyyppisenä, sen arvoa voidaan myös muokata. Vastakohtaisesti rvalue on lauseke, joka ilmaisee arvoa, tai lauseke, joka ilmaisee tilapäistä oliota, jota käyttäjä ei voi osoittaa ja jonka arvoa ei voi muokata. Tässä on yksinkertainen esimerkki:

```
int calc( int );

int main() {
    int lval, res;
```

```
    lval = 5; // lvalue: lval; rvalue: 5
    res = calc( lval );
        // lvalue: res;
        // rvalue: tilapäinen, ja sisältää calc()-funktion paluuarvon
    return 0;
}
```

Ensimmäisessä sijoituslausekkeessa lvalue on lval ja literaalivakio 5 on rvalue. Toisessa sijoituslausekkeessa lvalue on res ja tilapäinen muuttuja, joka sisältää calc()-funktion kutsun paluuarvon, on rvalue.

Joissakin tilanteissa käytetään lauseketta, joka on lvalue, kun odotetaan arvoa. Esimerkiksi:

```
int obj1;
int obj2;

int main() {
    // ...
    int local = obj1 + obj2;
    return 0;
}
```

obj1 ja obj2 ovat lvalue-lausekkeita. Kuitenkin yhteenlasku main()-funktiossa tarvitsee vain arvoja, jotka on tallennettu muuttujiin obj1 ja obj2. Ennen kuin yhteenlasku suoritetaan, haetaan arvot muuttujista obj1 ja obj2. Tätä toimenpidettä, kun haetaan arvo oliolle, jota lvalue-lauseke edustaa, kutsutaan lvalue rvalueksi -konversioksi.

Kun funktio odottaa arvona välitettävää argumenttia, tehdään lvalue rvalueksi -konversio silloin, kun argumentti on lvalue. Esimerkiksi:

```
#include <string>
string color( "purple" );
void print( string );

int main() {
    print( color ); // vastaa täsmälleen: lvalue rvalueksi -konversio
    return 0;
}
```

Koska argumentti välitetään arvona funktion print(color) kutsussa, lvalue rvalueksi -konversio tapahtuu siten, että arvo haetaan color-oliosta ja välitetään print(string)-funktiolle. Vaikka lvalue rvalueksi -konversio tapahtuu, vastaa color-argumentti siitä huolimatta täsmälleen print(string)-funktion kutsua.

Kaikki funktion kutsut eivät vaadi, että lvalue rvalueksi -konversio tapahtuu argumenteille. Viittaus edustaa lvalueta; kun funktiolla on viittausparametri, funktio, jota kutsutaan, vastaanottaa lvaluen. Tästä syystä ei lvalue rvalueksi -konversiota käytetä argumentille, jota vastaa viittausparametri. Olkoon esimerkiksi seuraava funktio:

```
#include <list>
void print( list<int> & );
```

Tällöin li edustaa seuraavassa kutsussa lvalueta, joka edustaa list<int>-oliota, joka välitetään print()-funktiolle.

```
list<int> li(20);

int main() {
    // ...
    print( li ); // vastaa täsmälleen: ei lvalue rvalueksi -konversiota
    return 0;
}
```

Kun li sidotaan viittausparametriin, se edustaa täsmällistä vastaavuutta.

Toinen konversio, joka sallitaan täsmällisen vastaavuuden tilanteessa, on taulukko osoittimeksi -konversio. Kuten mainitsimme kohdassa 7.3, funktion parametri ei ole koskaan taulukkotyyppinen. Sen sijaan parametri muunnetaan osoittimeksi taulukon ensimmäiseen elementtiin. Samalla tavalla argumentti, jonka tyyppi on taulukko N T (jossa N on taulukon elementtien lukumäärä ja T on taulukon elementtien tyyppi), konvertoidaan osoittimeksi tyyppiin T. Tämän argumentin tyyppikonversio on taulukko osoittimeksi -konversio. Vaikka konversio tapahtuu, nähdään argumentin siitä huolimatta vastaavan täsmälleen parametria osoitin tyyppiin T. Esimerkiksi:

```
int ai[3];
void putValues(int *);

int main() {
    // ...
    putValues(ai); // vastaa täsmälleen: taulukko osoittimeksi -konversio
    return 0;
}
```

Ennen kuin funktiota putValues() kutsutaan, tapahtuu taulukko osoittimeksi -konversio, joka konvertoi kolmen int-elementin taulukon argumentin osoittimeksi int-tyyppiin. Vaikka putValues()-funktiolla on osoitinparametri ja vaikka taulukko osoittimeksi -konversio tapahtuu argumentille, vastaa argumentti siitä huolimatta täsmälleen putValues()-funktion kutsua.

Seuraava konversio, joka sallitaan täsmällisen vastaavuuden tilanteessa, on funktio osoittimeksi -konversio. Tämä konversio esiteltiin lyhyesti kohdassa 7.9. Aivan kuten taulukkotyypin parametrilla, muunnetaan funktiotyyppinen parametri automaattisesti osoittimeksi funktioon. Funktiotyyppien argumentti konvertoidaan myös automaattisesti osoittimeksi funktioon. Tätä konversiota argumentin tyypille kutsutaan funktio osoittimeksi -konversioksi. Vaikka tämä konversio tapahtuu, nähdään argumentin siitä huolimatta vastaavan täsmälleen parametrityyppiä osoitin funktioon. Esimerkiksi:

```
int lexicoCompare( const string &, const string & );

typedef int (*PFI)( const string &, const string & );
void sort( string *, string *, PFI );
```

```
string as[10];

int main()
{
    // ...
    sort( as,
        as + sizeof(as)/sizeof(as[0] - 1),
        lexicoCompare // vastaa täsmälleen:
                // funktio osoittimeksi -konversio
      );

    return 0;
}
```

Ennen kuin sort()-funktiota kutsutaan, tapahtuu funktio osoittimeksi -konversio, joka konvertoi lexicoCompare-argumentin funktiotyypistä osoittimeksi funktiotyyppiin. Vaikka funktio odottaa osoitinta ja argumentti on funktion nimi ja vaikka funktio osoittimeksi -konversio tapahtuu, argumentti vastaa täsmälleen sort()-funktion kolmatta parametria.

Loput konversiot, joiden mainitaan kuuluvan täsmällisen vastaavuuden kategoriaan, ovat määrekonversioita. Määrekonversio vaikuttaa vain osoittimiin. Se on konversio, joka lisää const- tai volatile-määreen (tai molemmat) tyyppiin, johon osoitin osoittaa. Esimerkiksi:

```
int a[5] = { 4454, 7864, 92, 421, 938 };
int *pi = a;
bool is_equal( const int * , const int * );

void func( int *parm ) {

    // pi ja parm vastaavat täsmälleen: määrekonversiot
    if ( is_equal( pi, parm ) )
        // ...
    return 0;
}
```

Ennen kuin is_equal()-funktiota kutsutaan, konvertoidaan pi- ja parm-argumentti osoittimesta int-tyyppiin osoittimeksi const int -tyyppiin. Tämä konversio, joka lisää const-määreen tyyppiin, johon osoitin osoittaa, on määrekonversio. Vaikka funktio odottaa kahta osoitinta const int -tyyppiin ja argumentit ovat osoittimia int-tyyppiin, vastaavat argumentit täsmälleen is_equal()-funktion parametreja.

Määrekonversiota käytetään vain tyyppiin, jota osoitin osoittaa. Tyyppikonversiota ei käytetä, kun parametri on tyyppiä const tai volatile eikä argumentti itse ole:

```
extern void takeCI( const int );

int main() {
    int ii = ...;
    takeCI(ii); // ei määrekonversiota
    return 0;
```

```
}
```

Kun kutsutaan takeCI()-funktiota ja vaikka parametrin tyyppi on const int, ei määrekonversioita käytetä int-tyyppiseen ii-argumenttiin. Argumentti vastaa täsmälleen funktion parametria.

Näin on myös asianlaita, jos argumentti on osoitin ja const- tai volatile-määre käy itse osoittimeen:

```
extern void init( int *const );
extern int *pi;

int main() {
    // ...
    init(pi); // ei määrekonversiota
    return 0;
}
```

init()-funktion parametrin const-määrettä käytetään itse osoittimeen eikä tyyppiin, johon osoitin osoittaa. Tästä syystä kääntäjä ei huomioi const-määrettä, kun se harkitsee konversioiden käyttöä argumentille. Määrekonversiota ei tehdä argumentille pi, ja se vastaa täsmälleen funktion parametrin tyyppiä.

Kolmea ensimmäistä konversiota täsmällisen vastaavuuden kategoriassa (lvalue rvalueksi-, taulukko osoittimeksi- ja funktio osoittimeksi -konversiot) sanotaan usein *lvalue-muunnoksiksi*. Kuten tulemme näkemään kohdassa 9.4, vaikka lvalue-muunnokset ja määrekonversiot kuuluvat täsmällisen vastaavuuden kategoriaan, silloin, kun tarvitaan vain lvalue-muunnos, sitä pidetään arvojärjestyksessä parempana kuin määrekonversiota. Näemme tästä lisää yksityiskohtia seuraavassa kohdassa.

Täsmälliseen vastaavuuteen voidaan pakottaa käyttämällä eksplisiittistä tyyppimuunnosta. Olkoot esimerkiksi seuraavat ylikuormitetut funktiot:

```
extern void ff(int);
extern void ff(void *);
```

Silloin kutsu

```
ff( 0xffbc ); // calls ff(int)
```

vastaa ff(int)-funktiota täsmälleen, koska 0xffbc on int-tyyppinen literaalivakio kirjoitettuna heksadesimaalisella ilmaisulla. Ohjelmoija voi pakottaa ff(void*)-funktion käynnistyksen tekemällä eksplisiittisen tyyppikonversion kuten seuraavassa:

```
ff( reinterpret_cast<void *>(0xffbc) ); // kutsuu funktiota ff(void*)
```

Kun eksplisiittistä konversiota käytetään argumentille, argumentin tyypistä tulee tyyppimuunnoksen tulos. Eksplisiittisen tyyppimuunnoksen käyttö voi auttaa funktion ylikuormituksen ratkaisun ohjaamista. Jos esimerkiksi funktion ylikuormituksen ratkaisu on moniselitteinen, koska argumentit vastaavat yhtä hyvin kahta tai useampaa elinkelpoista funktiota, voidaan eksplisiittistä tyyppimuunnosta käyttää katkaisemaan moniselitteisyys ja pakottaa kutsu ratkaisemaan tietty elinkelpoinen funktio.

9.3.2 Ylennyksen yksityiskohdat

Ylennys (*promotion*) on jokin seuraavista konversioista.

- Argumentti, joka on char-, unsigned char- tai short-tyyppinen, ylennetään int-tyypiksi. Argumentti, joka on unsigned short -tyyppinen, ylennetään int-tyypiksi, jos koneen int-tyypin koko on suurempi kuin short-tyyppinen kokonaisluku; muussa tapauksessa se ylennetään unsigned int -tyypiksi.
- Argumentti, joka on float-tyyppinen, ylennetään double-tyypiksi.
- Argumentti, joka on lueteltu tyyppi, ylennetään ensimmäiseen seuraavista tyypeistä, joka voi edustaa kaikkia luetellun joukon vakioarvoja: int, unsigned int, long tai unsigned long.
- Argumentti, joka on bool-tyyppinen, ylennetään int-tyypiksi.

Ylennystä käytetään, kun argumentin tyyppi on jokin juuri kuvatuista tyypeistä ja funktion parametrin tyyppi on ylennetty tyyppi. Esimerkiksi:

```
extern void manip( int );

int main() {
    manip( 'a' ); // char-tyyppi ylennetään int-tyypiksi
    return 0;
}
```

Merkkiliteraali on char-tyyppi. Se ylennetään int-tyypiksi. Koska ylennetty tyyppi vastaa manip()-funktion parametrin tyyppiä, sanomme, että funktion kutsu vaatii ylennyksen argumentilleen.

Oletetaan, että on seuraava esimerkki:

```
extern void print( unsigned int );
extern void print( int );
extern void print( char );

unsigned char uc;
print( uc ); // print( int ): vain uc vaatii ylennyksen
```

Koneessa, jossa unsigned char -tyyppi vie yhden tavun ja int-tyyppi vie neljä tavua muistia, tekee ylennys unsigned char -tyypistä int-tyypin, koska siinä arkkitehtuurissa int-tyyppi voi esittää kaikki unsigned char -tyypin arvot. Edellisten ylikuormitettujen funktioiden esittelyiden ja kuvatun arkkitehtuurin mukaan funktio, joka parhaiten vastaa unsigned char -tyyppistä argumenttia, on funktio print(int). Kahden muun funktion täsmäämiseksi vaadittaisiin vakiokonversiota.

Seuraava esimerkki kuvaa luetellun argumenttityypin ylennystä:

```
enum Stat { Fail, Pass };

extern void ff( int );
extern void ff( char );
```

```
int main() {
    // ok: lueteltu vakio Pass ylennetään int-tyypiksi
    ff( Pass );  // ff( int )
    ff( 0 );     // ff( int )
    return 0;
}
```

Lueteltujen tyyppien ylennykset voivat joskus olla yllättäviä. Toteutukset valitsevat usein luetellun tyypin esitystavan lueteltujen vakioiden perusteella. Oletetaan esimerkiksi, että meillä on aikaisemmin kuvattu arkkitehtuuri (char-tyypit vievät yhden tavun, int-tyypit neljä tavua) ja seuraava lueteltu tyyppi:

```
enum e1 { a1, b1, c1 };
```

Koska on olemassa vain kolme lueteltua vakiota — a1, b1 ja c1 — joiden arvot ovat vastaavasti 0, 1 ja 2 ja koska kaikki tämän luetellun tyypin arvot voidaan esittää char-tyypillä, valitsee toteutus yleensä char-tyypin e1:n esitystavaksi. Oletetaan kuitenkin, että meillä on toinen lueteltu tyyppi e2, jossa on erilaiset luetellut vakiot:

```
enum e2 { a2, b2, c2=0x80000000 };
```

Koska luetelluissa vakioissa on arvo 0x80000000, toteutus on pakotettu valitsemaan e2:n esitystavalle tyypin, joka voi sisältää arvon 0x80000000. e2:lle valittu esitystapa on unsigned int.

Joten, vaikka sekä e1 ja e2 ovat lueteltuja tyyppejä, niiden esitystavat eroavat toisistaan. Tämä saa aikaan sen, että e1 ja e2 ylennetään eri tyypeiksi. Esimerkiksi:

```
#include <string>

string format( int );
string format( unsigned int );

int main() {
    format(a1); // kutsuu funktiota format( int )
    format(a2); // kutsuu funktiota format( unsigned int )
    return 0;
}
```

Koska format()-funktion ensimmäisessä kutsussa argumentin tyyppi e1 on char-tyypin esitystavassa, argumentti ylennetään int-tyypiksi ja funktio format(int) tulee valituksi kutsua varten. Koska format()-funktion toisessa kutsussa e2-argumentti on unsigned int -tyypin esitystavassa, argumentti ylennetään unsigned int -tyypiksi. Tämä saa aikaan sen, että funktio format(unsigned int) valitaan toista kutsua varten. Siitä syystä sinun tulisi olla tietoinen seuraavasta: kaksi lueteltua tyyppiä käyttäytyvät melko eri tavalla funktion ylikuormituksen ratkaisun aikana riippuen siitä, mitä arvoja niiden luetellut vakiot ovat, koska ne määräävät tyypin, johon ne ylennetään.

9.3.3 Vakiokonversion yksityiskohdat

Vakiokonversiosta on ryhmitelty viisi erilaista konversiokategoriaa:

1. Kokonaiskonversiot: minkä tahansa kokonaistyypin tai luetellun tyypin konversio mihin tahansa toiseen kokonaistyyppiin (poislukien konversiot, jotka mainittiin aikaisemmin ylennyksinä).
2. Liukulukukonversiot: minkä tahansa liukulukutyypin konversio mihin tahansa toiseen liukulukutyyppiin (poislukien konversiot, jotka mainittiin aikaisemmin ylennyksinä).
3. Liukuluku-kokonais-konversiot: minkä tahansa liukulukutyypin konversio mihin tahansa kokonaistyyppiin tai minkä tahansa kokonaistyypin mihin tahansa liukulukutyyppiin.
4. Osoitinkonversiot: kokonaisluvun nolla-arvon konversio osoitintyypiksi ja minkä tahansa osoitintyypin konversio `void*`-tyypiksi.
5. `bool`-konversiot: minkä tahansa kokonaistyypin, liukulukutyypin, luetellun joukon tyypin tai osoitintyypin konversio `bool`-tyypiksi.

Tässä on joitakin esimerkkejä:

```
extern void print( void* );
extern void print( double );

int main() {
    int i;
    print( i );  // vastaa funktiota print( double );
                 // i on muunnettu vakiokonversiolla
                 // kokonaisluvusta int double-tyypiksi
    print( &i ); // vastaa funktiota print( void* );
                 // &i on muunnettu vakiokonversiolla
                 // osoittimesta kokonaislukuun int* void*-tyypiksi
    return 0;
}
```

Konversiot, jotka on ryhmitetty kategorioihin 1, 2 ja 3, ovat potentiaalisesti vaarallisia konversioita, koska konversion kohdetyyppi ei voi edustaa kaikkia lähdetyypin edustamia arvoja. Esimerkiksi `float`-tyyppi ei voi esittää samalla tarkkuudella kaikkia niitä arvoja, joita voidaan esittää `int`-tyypillä. Tästä syystä näiden kategorioiden konversiot ovat vakiokonversioita eivätkä ylennyksiä.

```
int i;
void calc( float );
int main() {
    calc( i ); // liukuluku-kokonais-vakiokonversio
               // on mahdollisesti vaarallinen, koska se riippuu i:n arvosta
    return 0;
}
```

Kun calc()-funktiota kutsutaan, käytetään liukuluku-kokonais-konversiota argumentin muuntamiseen int-tyypistä float-tyyppiin. Riippuen arvosta, joka on tallennettu i-muuttujaan, on mahdollista, että arvoa i ei voi tallentaa float-tyyppiseen parametriin ilman tarkkuuden menetystä.

Kaikkien konversioiden katsotaan vaativan samanlaisen työn. Esimerkiksi konversio char-tyypistä unsigned char -tyyppiin ei tapahdu (ei sidota järjestystä) aikaisemmin kuin konversio char-tyypistä double-tyyppiin. Tyyppien läheisyyttä ei oteta huomioon. Jos kaksi elinkelpoista funktiota vaativat vakiokonversiot argumenteille, jotta ne vastaisivat parametriensa tyyppejä, kutsu on moniselitteinen ja saa aikaan käännöksenaikaisen virheen. Olkoon esimerkiksi seuraava pari ylikuormitettuja funktioita

```
extern void manip( long );
extern void manip( float );
```

Silloin seuraava kutsu on moniselitteinen:

```
int main() {
    manip( 3.14 ); // virhe: moniselitteinen;
                   // manip( float ) ei ole parempi
    return 0;
}
```

Literaalivakio 3.14 on double-tyyppinen. Vastaavuus saavutetaan jommallakummalla funktiolla vakiokonversion keinoin. Koska on olemassa kaksi vakiokonversiomahdollisuutta, kutsu saa aikaan käännöksenaikaisen virheen moniselitteisyydestään. Yksikään vakiokonversioista ei saa etuoikeuksia sidontajärjestyksessä suhteessa muihin. Ohjelmoija voi ratkaista moniselitteisyyden joko eksplisiittisellä tyyppimuunnoksella kuten tässä

```
manip( static_cast<long>( 3.14 ) ); // manip( long )
```

tai käyttämällä float-tyypin vakioloppuliitettä:

```
manip( 3.14F ); // manip( float )
```

Tässä on joitakin esimerkkejä funktioiden kutsuista, jotka ovat moniselitteisiä ja saavat aikaan käännöksenaikaisen virheen, koska ne vastaavat useampaa kuin yhtä funktiota ylikuormitettujen funktioiden joukosta:

```
extern void farith( unsigned int );
extern void farith( float );

int main() {
    // jokainen kutsu on moniselitteinen
    farith( 'a' );     // argumentin tyyppi on char
    farith( 0 );       // argumentin tyyppi on int
    farith( 2uL );     // argumentin tyyppi on unsigned long
    farith( 3.14159 ); // argumentin tyyppi on double
    farith( true );    // argumentin tyyppi on bool
    return 0;
}
```

Osoittimen vakiokonversio voi joskus näyttää vaikeasti ymmärrettävältä. Erityisesti arvo 0 voidaan muuntaa miksi tahansa osoitintyypiksi: tällä tavalla luodun osoittimen arvoa kutsutaan *osoittimen nolla-arvoksi*. Tämä 0-arvo voi olla mikä tahansa kokonaistyyppinen vakiolauseke. Esimerkiksi:

```
void set(int*);

int main() {
    // osoitinkonversio arvosta 0 tyyppiin int* käy molempiin argumentteihin
    set( 0L );
    set( 0x00 );
    return 0;
}
```

Vakiolausekkeet `0L` (arvo 0, jonka tyyppi on `long int`) ja `0x00` (heksadesimaalinen kokonaisluku, jonka arvo on 0) ovat kokonaistyyppejä ja voidaan siten konvertoida nolla-osoittimiksi, joiden tyyppi on `int*`.

Koska luetellun joukon tyypit eivät kuitenkaan ole kokonaislukutyyppejä, ei 0-arvoista tyyppiä voi konvertoida osoitintyypiksi. Esimerkiksi:

```
enum EN { zr = 0 };
set( zr ); // virhe: zr:n konversio int*-tyypiksi ei onnistu
```

Funktion `set()` kutsu on virheellinen, koska konversio ei ole mahdollinen luetellun joukon arvon `zr` ja parametrityypin `int*` välillä, vaikka luetellun joukon arvo olisi nolla.

Myös jotain muuta pitäisi huomioida: vakiolausekkeen 0 tyyppi on `int`. Vakiokonversio on välttämätön, jotta tämä vakiolauseke saataisiin osoitintyypiksi. Jos ylikuormitettujen funktioiden joukossa on funktio, jonka parametrin tyyppi on `int`, olisi tuo funktio parempi 0-argumentille. Esimerkiksi:

```
void print( int );
void print( void * );

void set( const char* );
void set( char* );

int main() {
    print( 0 ); // kutsuu funktiota print( int )
    set( 0 );   // moniselitteinen
    return 0;
}
```

Argumentti vastaa täsmälleen kutsua `print(int)`-funktioon. Kuitenkin, jos kutsuttaisiin `print(void*)`-funktiota, olisi vakiokonversio välttämätön 0-arvon saattamiseksi osoitintyypiksi. Koska täsmällinen vastaavuus on parempi kuin vakiokonversio, valitaan `print(int)`-funktio kutsua varten. Funktion `set()` kutsu on moniselitteinen, koska 0 vastaa molempien `set()`-funktioiden parametreja vakiokonversion soveltamisen jälkeen. Koska kumpikin funktio on yhtä hyvä kutsua varten, johtaa funktion `set()` kutsu moniselitteisyysvirheeseen.

Viimeinen osoitinkonversio mahdollistaa argumentin minkä tahansa osoitintyypin konvertoimisen parametrin void*-tyypiksi, koska void* on geneerinen osoitin tietotyyppiin, joka voi sisältää osoittimen osoittaman minkä tahansa tietotyypin arvon. Tässä on joitakin esimerkkejä:

```
#include <string>
extern void reset( void * );

void func( int *pi, string *ps ) {
    // ...
    reset( pi ); // osoitinkonversio: tyypistä int* void*-tyyppiin
    // ...
    reset( ps ); // osoitinkonversio: tyypistä string* void*-tyyppiin
    return 0;
}
```

Vain osoitin tietotyyppiin voidaan konvertoida void*-tyypiksi osoittimen vakiokonversiolla. Osoitinta funktioon ei voida konvertoida void*-tyypiksi vakiokonversiolla. Esimerkiksi:

```
typedef int (*PFV)();
extern PFV testCases[10]; // osoitintaulukko funktioihin

extern void reset( void * );

int main() {
    // ...
    reset( testCases[0] ); // virhe: ei vakiokonversiota
                    // funktion int(*)() ja tyypin void* välillä
    return 0;
}
```

9.3.4 Viittaukset

Funktion kutsun argumentti tai funktion parametri voi olla viittaus. Kuinka viittaus vaikuttaa tyyppikonversioiden sääntöihin?

Tutkikaamme ensiksi, mitä tapahtuu, jos argumentti on viittaus. Argumentin tyyppi ei koskaan ole viittaustyyppi. Kun argumentti on viittaus, se on lvalue, jonka tyyppi on olion tyyppi, johon se viittaa. Mietitäänpä tätä esimerkkiä:

```
int i;
int& ri = i;
void print( int );

int main() {
    print( i );  // lvalue-argumentti, jonka tyyppi on int
    print( ri ); // samoin tässä
    return 0;
}
```

Argumentin tyyppi molemmissa funktion kutsuissa on int. Tosiasialla, että toisessa kutsussa käytetään viittausta argumenttina, ei ole vaikutusta argumentin tyyppiin.

Vakiokonversiot ja ylennykset, joita harkitaan, kun argumentti on viittaus T-tyyppiin, ovat samoja kuin ne, joita harkitaan, kun argumentti on T-tyyppinen olio. Esimerkiksi:

```
int i;
int& ri = i;
void calc( double );

int main() {
    calc( i );  // liukuluku-kokonais-vakiokonversio
    calc( ri ); // samoin tässä
    return 0;
}
```

Nyt kysymme, kuinka viittausparametrit vaikuttavat argumenttiin sovellettuihin konversioihin? Argumentin täsmäämisen tulos viittausparametrin kanssa on jokin seuraavista mahdollisuuksista:

1. Argumentti on sopiva alustaja viittausparametrille. Tässä tapauksessa sanomme, että argumentti vastaa täysin parametria. Esimerkiksi:

```
void swap( int &, int & );

void manip( int i1, int i2 ) {
    // ...
    swap( i1, i2 ); // ok: kutsuu funktiota swap( int &, int & )
    // ...
    return 0;
}
```

2. Argumentti ei voi alustaa viittausparametria. Tässä tapauksessa vastaavuustilannetta ei ole eikä argumenttia voida käyttää funktion kutsumiseen. Esimerkiksi:

```
int obj;
void frd( double & );
int main() {
    frd( obj ); // virhe: parametrin pitää olla: const double &
    return 0;
}
```

   Funktion frd() kutsu on virheellinen. Argumentti on int-tyyppinen ja se pitäisi konvertoida double-tyyppiseksi, jotta se vastaisi viittausparametria. Tämän konversion tulos on tilapäinen. Koska viittaus ei ole const-tyyppiin, ei tilapäistä arvoa voida käyttää viittauksen alustamiseen.

   Tässä on toinen esimerkki, jossa vastaavuutta ei löydy viittausparametrin ja argumentin välillä:

```
class B;
void takeB( B& );
B giveB();

int main() {
    takeB( giveB() ); // virhe: parametrin pitää olla: const B &
    return 0;
}
```

   Funktion takeB() kutsu on virheellinen. Argumentti on funktion kutsun paluuarvo; se on tilapäinen arvo, jota ei voida käyttää sellaisen viittauksen alustamiseen, joka ei osoita const-tyyppiin.

   Jos molemmissa tapauksissa viittausparametri olisi viittaus const-tyyppiin, vastaisi argumentti täsmälleen parametria.

Tässä tulisi huomioida, että sekä lvalue rvalueksi -konversion että viittauksen alustamisen katsotaan kuuluvan täsmälliseen vastaavuuteen. Esimerkiksi seuraavassa koodikatkelmassa

```
void print( int );
void print( int& );

int iobj;
int &ri = iobj;

int main() {
    print( iobj ); // virhe: moniselitteinen
    print( ri );   // virhe: moniselitteinen
    print( 86 );   // ok: kutsuu funktiota print( int )
    return 0;
}
```

ensimmäinen funktion kutsu on virheellinen. Olio `iobj` on argumentti, joka vastaa täsmälleen molempia `print()`-funktioita ja saa aikaan sen, että funktion kutsu on moniselitteinen. Tämä pätee myös toiseen funktion kutsuun. Viittaus `ri` kohdistuu olioon, joka vastaa täsmälleen molempia `print()`-funktioita. Kolmas kutsu on kuitenkin tässä OK. Funktio `print(int&)` ei ole elinkelpoinen funktio tälle kutsulle. Kokonaislukuvakio on rvalue eikä se ole kelvollinen `const`-määreettömän viittausparametrin alustajaksi. Ainoa funktio elinkelpoisten funktioiden joukosta kutsulle `print(86)` on `print(int)`. Koska se on ainoa elinkelpoinen funktio, se valitaan kutsua varten.

Kiteytettynä: kun on kyse viittausparametrista, argumentti joko vastaa täsmälleen, jos argumentti on kelvollinen viittauksen alustaja, tai vastaavuutta ei ole, ellei argumentti ole kelvollinen viittauksen alustaja.

Harjoitus 9.6

Nimeä kaksi pienempää konversiota, jotka sallitaan täsmällisessä vastaavuudessa.

Harjoitus 9.7

Millainen on seuraavien funktiokutsujen argumenttien konversioiden paremmuusjärjestys?

```
(a) void print( int *, int );
    int arr[6];
    print( arr, 6 ); // funktion kutsu

(b) void manip( int, int );
    manip( 'a', 'z' ); // funktion kutsu

(c) int calc( int, int );
    double dobj;
    double = calc( 55.4, dobj ); // funktion kutsu

(d) void set( const int * );
    int *pi;
    set( pi ); // funktion kutsu
```

Harjoitus 9.8

Mitkä seuraavista funktioiden kutsuista, jos yksikään, ovat virheellisiä, koska konversiota ei ole olemassa argumentin ja funktion parametrin tyypin välillä?

```
(a) enum Stat { Fail, Pass };
    void test( Stat );
    test( 0 ); // funktion kutsu

(b) void reset( void * );
    reset( 0 ); // funktion kutsu

(c) void set( void * );
    int *pi;
    set( pi ); // funktion kutsu

(d) #include <list>
    list<int> oper();
    void print( list<int> & );
    print( oper() ); // funktion kutsu

(e) void print( const int );
    int iobj;
    print( iobj ); // funktion kutsu
```

9.4 Funktion ylikuormituksen ratkaisun yksityiskohdat ◆

Kuten selitimme kohdassa 9.2, funktion ylikuormituksen ratkaisussa on kolme vaihetta. Näistä vaiheista voidaan tehdä yhteenveto seuraavasti:

1. Yksilöi funktioehdokkaat, joita voidaan harkita kutsua varten, ja kutsun argumenttiluettelon ominaisuudet.
2. Valitse elinkelpoiset funktiot ehdokkaista: valitse funktiot, joita voidaan kutsua kutsussa määritetyin argumentein ottaen huomioon niiden lukumäärät ja tyypit.
3. Valitse funktio, joka parhaiten vastaa kutsua, laittamalla konversiot paremmuusjärjestykseen, joita argumentteihin käytetään, jotta ne vastaisivat elinkelpoisten funktioiden parametrien tyyppejä.

Olemme nyt valmiita tutkimaan näitä kolmea vaihetta tarkemmin.

9.4.1 Funktioehdokkaat

Ehdokasfunktio on sellainen, jonka nimi on sama kuin kutsutun funktion nimi. Ehdokasfunktio löytyy jollakin seuraavista tavoista:

1. Funktion esittely on näkyvissä kutsupaikassa. Seuraavassa esimerkissä

   ```
   void f();
   ```

```
void f( int );
void f( double, double = 3.4 );
void f( char*, char* );

int main() {
    f( 5.6 ); // tälle kutsulle on neljä funktioehdokasta
    return 0;
}
```

funktioita f() on esiteltynä neljä ilmentymää globaalilla viittausalueella ja ne näkyvät kutsupaikassa. Ne ovat siten osa funktioehdokkaista.

2. Jos funktion argumentin tyyppi on esiteltynä nimiavaruudessa, sen jäsenfunktiot, joilla on sama nimi kuin kutsutulla funktiolla, lisätään funktioehdokkaisiin. Esimerkiksi:

```
namespace NS {
    class C { /* ... */ };
    void takeC( C& );
}

// cobj:n tyyppi on luokka C, joka on esitelty nimiavaruudessa NS
NS::C cobj;

int main() {
            // ei yhtäkään takeC()-funktiota näkyvissä kutsupaikassa
    takeC( cobj ); // ok: kutsuu funktiota NS::takeC( C& )
            // koska argumentti on tyyppiä NS::C
            // harkitaan funktiota takeC(), joka on esitelty
            // nimiavaruudessa NS
    return 0;
}
```

Funktioehdokkaat ovat siten niiden funktioiden yhdistelmä, jotka ovat näkyvissä kutsuhetkellä. Ne ovat myös funktioita, jotka on esitelty argumenttityyppien nimiavaruuksissa.

Kun yksilöimme ylikuormitettuja funktioita, jotka ovat näkyvissä kutsuhetkellä, aikaisemmin näkemämme säännöt ylikuormitetun funktiojoukon rakentamiseksi käyvät yhä.

Funktio, joka on esitelty sisäkkäisellä viittausalueella, piilottaa pikemminkin kuin ylikuormittaa samannimisen funktion, joka on sitä ympäröivällä viittausalueella. Sellaisessa tilanteessa funktioehdokkaat ovat niitä, jotka on esitelty sisäkkäisellä viittausalueella; tarkoittaa funktioita, jotka eivät peity funktion kutsussa. Seuraavassa esimerkissä funktioehdokkaat, jotka ovat näkyviä kutsuhetkellä, ovat format(double) ja format(char*):

```
char* format( int );
void g() {
    char* format( double );
    char* format( char* );
    format(3); // kutsuu funktiota format( double )
}
```

Koska globaalilla viittausalueella esitelty format(int)-funktio jää piiloon, sitä ei oteta mukaan funktioehdokkaisiin.

Funktioehdokkaat voidaan tuoda esille käyttämällä esittelyitä, jotka ovat näkyviä kutsuhetkellä. Mietitäänpä seuraavaa esimerkkiä:

```
namespace libs_R_us {
    int max( int, int );
    double max( double, double );
}

char max( char, char );

void func()
{
    // nimiavaruuden funktiot eivät ole näkyviä
    // nämä kolme kutsua kutsuvat globaalia funktiota max( char, char )
    max( 87, 65 );
    max( 35.5, 76.6);
    max( 'J', 'L' );
}
```

max()-funktiot, jotka on määritelty nimiavaruudessa libs_R_us, eivät ole näkyvissä kutsuhetkellä. Ainoa näkyvissä oleva funktio on max(), joka on esitelty globaalilla viittausalueella. Tämä funktio on ainoa funktioehdokkaiden joukossa; se on funktio, jota kaikki kolme kutsua kutsuvat funktiossa func(). Voimme käyttää using-esittelyä saadaksemme libs_R_us-nimiavaruudessa esitellyt max()-funktiot näkyviin. Mihin laittaisimme using-esittelyn? Jos sijoitamme sen globaalille viittausalueelle

```
char max( char, char );
using libs_R_us::max; // using-esittely
```

lisätään libs_R_us-nimiavaruuden max()-funktiot ylikuormitettuihin funktioihin, johon kuuluu globaalilla viittausalueella esitelty max()-funktio. Kaikki kolme funktiota ovat nyt näkyviä func()-funktiossa ja niistä tulee osa funktioehdokkaista. Kaikki kolme funktiota, jotka ovat näkyvissä kutsuhetkellä, ratkaistaan func()-funktiossa seuraavasti:

```
void func()
{
    max( 87, 65 );     // kutsuu funktiota libs_R_us::max( int, int )
    max( 35.5, 76.6 ); // kutsuu funktiota libs_R_us::max( double, double )
    max( 'J', 'L' );   // kutsuu funktiota ::max( char, char )
}
```

Entä sitten, jos olisimme sijoittaneet using-esittelyn func()-funktion paikalliselle viittausalueelle kuten seuraavassa?

```
void func()
{
    // using-esittely
    using libs_R_us::max;

    // funktion kutsut kuten edellä
}
```

Mitkä max()-funktiot tulisi ottaa mukaan funktioehdokkaisiin? Muista, että using-esittelyt ovat sisäkkäin. Koska using-esittely on paikallisella viittausalueella, globaali max(char, char)-funktio jää piiloon. Ainoat näkyvät funktiot kutsuhetkellä ovat

```
libs_R_us::max( int, int )
libs_R_us::max( double, double )
```

Nämä kaksi funktiota ovat ainoat, jotka ovat osa funktioehdokkaita. Kutsut func()-funktiossa ratkaistaan sitten seuraavasti:

```
void func()
{
    // using-esittely
    // globaali max( char, char )-funktio peittyy
    using libs_R_us::max;

    max( 87, 65 );     // kutsuu funktiota libs_R_us::max( int, int )
    max( 35.5, 76.6 ); // kutsuu funktiota libs_R_us::max( double, double )
    max( 'J', 'L' );   // kutsuu funktiota libs_R_us::max( int, int )
}
```

Myös using-direktiivit vaikuttavat siihen, kuinka muodostamme funktioehdokkaat. Oletetaan, että päätämme käyttää using-direktiiviä using-esittelyn sijasta saadaksemme libs_R_us-nimiavaruuden max()-funktiot näkyviin func()-funktiossa. Kun esimerkiksi seuraava using-direktiivi on globaalilla viittausalueella, funktioehdokkaisiin kuuluvat globaali max(char, char)-funktio ja max(int, int)- sekä max(double, double)-funktiot, jotka on esitelty libs_R_us-nimiavaruudessa:

```
namespace libs_R_us {
    int max( int, int );
    double max( double, double );
}

char max( char, char );
using namespace libs_R_us; // using-direktiivi

void func()
{
    max( 87, 65 );     // kutsuu funktiota libs_R_us::max( int, int )
    max( 35.5, 76.6 ); // kutsuu funktiota libs_R_us::max( double, double )
    max( 'J', 'L' );   // kutsuu funktiota ::max( char, char )
}
```

Mitä tapahtuu, jos using-direktiivi sijoitetaankin func()-funktion paikalliselle viittausalueelle kuten seuraavassa?

```
void func()
{
    // using-direktiivi
    using namespace libs_R_us;

    // funktion kutsut kuten edellä
}
```

Mitkä max()-funktiot ovat osa ehdokasfunktioita? Muista, että using-direktiivi saa nimiavaruuden jäsenet näkyviin aivan kuin ne olisi esitelty nimiavaruuden ulkopuolella paikassa, jossa nimiavaruus on määritelty. Esimerkissämme libs_R_us-nimiavaruuden jäsenet ovat näkyvissä func()-funktion paikallisella viittausalueella aivan kuin ne olisi esitelty nimiavaruutensa ulkopuolella, globaalilla viittausalueella. Tämä saa aikaan sen, että ylikuormitettujen funktioiden joukko, joka on näkyvissä func()-funktiossa, on sama kuin aikaisemmin ja sisältää seuraavat funktiot:

```
max( char, char )
libs_R_us::max( int, int )
libs_R_us::max( double, double )
```

func()-funktion kutsujen ratkaisuun ei vaikuta, esiintyykö using-direktiivi globaalilla viittausalueella vai func()-funktion paikallisella viittausalueella:

```
void func()
{
    using namespace libs_R_us;

    max( 87, 65 );     // kutsuu funktiota libs_R_us::max( int, int )
    max( 35.5, 76.6 ); // kutsuu funktiota libs_R_us::max( double, double )
    max( 'J', 'L' );   // kutsuu funktiota ::max( char, char )
}
```

Funktioehdokkaat ovat siten niiden funktioiden yhdistelmä, jotka ovat näkyvissä kutsuhetkellä — mukaan lukien funktiot, jotka on esitelty using-esittelyillä ja using-direktiiveillä — ja jäsenfunktiot, jotka on esitelty nimiavaruuksissa, jotka liittyvät argumenttien tyyppeihin. Esimerkiksi:

```
namespace basicLib {
    int print( int );
    double print( double );
}
namespace matrixLib {
    class matrix { /* ... */ };
    void print( const matrix & );
}
```

```
void display()
{
    using basicLib::print;

    matrixLib::matrix mObj;
    print( mObj ); // kutsuu funktiota matrixLib::print( const matrix& )

    print( 87 );   // kutsuu funktiota basicLib::print( int )
}
```

Mitkä ovat funktioehdokkaita kutsulle print(mObj)? Funktiot basicLib::print(int) ja basicLib::print(double), jotka on esitelty using-esittelyllä display()-funktiossa, ovat ehdokasfunktioita, koska ne ovat näkyvissä kutsuhetkellä. Koska funktion kutsun argumentti on tyyppiä matrixLib::matrix, funktio print(), joka on esitelty matrixLib-nimiavaruudessa, on myös funktioehdokas. Mitkä ovat funktioehdokkaita kutsulle print(87)? Vain funktiot basicLib::print(int) ja basicLib::print(double), jotka ovat näkyvissä kutsuhetkellä, ovat funktioehdokkaita. Koska argumentin tyyppi on int, ei muita nimiavaruuksia tutkita lisäehdokkaiden löytämiseksi.

9.4.2 Elinkelpoiset funktiot

Elinkelpoinen funktio on eräs funktioehdokkaista. Elinkelpoisen funktion parametriluettelossa on yhtä monta tai useampia parametreja kuin kutsussa on argumentteja. Jälkimmäisessä tapauksessa käytetään oletusargumentteja lisäparametreille niin, että funktiota voidaan kutsua argumenttiluettelon mukaisella argumenttimäärällä. Elinkelpoinen funktio on sellainen, jolle on olemassa konversioita, joilla jokainen argumenttityyppi voidaan muuntaa elinkelpoisen funktion parametriluettelon vastaavan parametrin mukaiseksi. Harkittavat konversiot ovat niitä, jotka esiteltiin kohdassa 9.3.

Seuraavassa esimerkissä on kaksi elinkelpoista funktiota kutsulle f(5.6): ne ovat f(int) ja f(double).

```
void f();
void f( int );
void f( double );
void f( char*, char* );

int main() {
    f( 5.6 ); // 2 elinkelpoista funktiota: f( int ) ja f( double )
    return 0;
}
```

f(int) on elinkelpoinen funktio, koska sillä on vain yksi parametri. Tämä vastaa funktion kutsun argumenttien lukumäärää. On myös olemassa vakiokonversio argumentin tyypistä double tyypiksi int. f(double) on myös elinkelpoinen funktio. Tällä elinkelpoisella funktiolla on vain yksi parametri ja sen tyyppi on double, joka vastaa täysin kutsun argumenttia. Funktioehdokkaat f() ja f(char*,char*) jätetään pois elinkelpoisten funktioiden joukosta, koska näitä funktioita ei voi kutsua yhdellä argumentilla.

Seuraavassa esimerkissä on ainoa elinkelpoinen funktio kutsulle format(3) funktio format(double). Vaikka funktioehdokasta format(char*) voidaan kutsua yhdellä argumentilla, ei konversiota ole olemassa argumenttityypin int ja parametrityypin char* välille. Koska tyyppikonversiota ei ole, jätetään tämä funktio pois elinkelpoisten funktioiden joukosta.

```
char* format( int );
void g() {
    // globaali funktio format( int ) peittyy
    char* format( double );
    char* format( char* );
    format(3); // ainoa elinkelpoinen funktio: format( double )
}
```

Seuraavassa esimerkissä kaikki kolme funktioehdokasta ovat elinkelpoisten funktioiden joukossa, kun kutsutaan max()-funktiota func()-funktiossa. Kaikkia kolmea funktiota voidaan kutsua kahdella argumentilla. Koska argumentit ovat int-tyyppisiä, ne vastaavat täysin funktion libs_R_us::max(int,int) parametreja. Ne voidaan konvertoida libs_R_us::max(double,double)-funktion parametreiksi liukuluku-kokonais-vakiokonversiolla ja ne voidaan konvertoida max(char,char)-funktion parametreiksi kokonais-vakiokonversiolla.

```
namespace libs_R_us {
    int max( int, int );
    double max( double, double );
}

// using-esittely
using libs_R_us::max;

char max( char, char );

void func()
{
    // kolme max()-funktiota ovat elinkelpoisia funktioita
    max( 87, 65 ); // kutsuu funktiota libs_R_us::max( int, int )
}
```

Huomaa, että funktioehdokas, jolla on useita parametreja, jätetään pois elinkelpoisten funktioiden joukosta heti, kun jotain funktion kutsun argumenteista ei voida konvertoida elinkelpoisen funktion parametriluettelon vastaavaksi parametriksi. Tämä pätee, vaikka olisi olemassa konversioita kaikille muille argumenteille. Seuraavassa esimerkissä min(char*,int)-funktio jätetään pois elinkelpoisten funktioiden joukosta, koska konversiota ei ole olemassa ensimmäisen argumentin int-tyypin ja vastaavan funktion parametrin char*-tyypin välille. Vaikka toinen argumentti vastaa täysin funktion toista argumenttia, jätetään funktio siitä huolimatta pois:

```
extern double min( double, double );
extern int min( char*, int );
```

```
void func()
{
    // yksi funktioehdokas: min( double, double )
    min( 87, 65 ); // kutsuu funktiota min( double, double )
}
```

Ellei funktioehdokkaiden eliminoimisen aikana löydy elinkelpoisia funktioita, joiden parametrien lukumäärä ei täsmää tai joille ei löydy sopivia tyyppikonversioita, kutsu johtaa käännöksenaikaiseen virheeseen. Sanomme tässä tapauksessa, että vastaavuutta ei löydy.

```
void print( unsigned int );
void print( char* );
void print( char );

int *ip;
class SmallInt { /* ... */ };
SmallInt si;

int main() {
    print( ip ); // virhe: ei elinkelpoista funktiota: ei vastaavuutta
    print( si ); // virhe: ei elinkelpoista funktiota: ei vastaavuutta
    return 0;
}
```

9.4.3 Parhaiten elinkelpoinen funktio

Parhaiten elinkelpoinen funktio on sellainen, jonka parametrit parhaiten vastaavat argumenttien tyyppejä. Jokaista funktiota varten laitetaan jokaisen argumentin tyyppikonversiot paremmuusjärjestykseen, jotta voitaisiin päätellä, kuinka jokainen argumentti täsmää vastaavaan parametriin. (Kohdassa 9.2 kuvataan tuettuja tyyppikonversioita.) Parhaiten elinkelpoinen funktio on sellainen, johon pätevät molemmat seuraavat asiat.

1. Argumentteihin kohdistuneet konversiot eivät ole *huonompia* kuin konversiot, jotka ovat tarpeen kutsuttaessa mitä tahansa muuta elinkelpoista funktiota.
2. Joidenkin argumenttien konversiot ovat *parempia* kuin konversiot, jotka ovat tarpeen samoille argumenteille, kun kutsutaan mitä tahansa muuta elinkelpoista funktiota.

Voi olla, että tyyppikonversioita tehdään enemmän kuin yksi, jotta argumentti saataisiin vastaavaksi funktion parametrin tyypiksi. Esimerkiksi seuraavassa koodiesimerkissä

```
int arr[3];
void putValues(const int *);

int main() {
    putValues(arr); // 2 konversiota järjestyksessä:
                    // taulukko osoittimeksi + määreen konversio
    return 0;
}
```

konversion järjestys tapahtuu konvertoimalla arr-argumentti kolmen int-tyypin taulukosta osoitintyypiksi const int. Tämä konversiosarja muodostuu seuraavista konversioista:

1. Taulukko osoittimeksi -konversio muuntaa kolmen int-tyypin taulukkoargumentin osoittimeksi int-tyyppiin.
2. Määreen konversio muuntaa osoittimen int-tyypiin osoittimeksi tyyppin const int.

Tämän takia on sopivampaa sanoa, että argumentin konvertoimisessa elinkelpoisen funktion parametrin tyypiksi käytetään *konversiosarjaa*. Koska argumentin konversiossa vastaavaksi parametrin tyypiksi käytetään konversiosarjaa yksittäisen konversion sijasta, laittaa funktion ylikuormituksen ratkaisun kolmas vaihe tästä syystä ne paremmuusjärjestykseen.

Konversiosarja on paremmuusjärjestyksen huonoimpia. Kuten kohdassa 9.2 kuvattiin, konversiot laitetaan paremmuusjärjestykseen seuraavasti: täydellinen vastaavuus on parempi kuin ylennys ja ylennys on parempi kuin vakiokonversio. Edellisessä esimerkissä molemmat konversiot täsmäävät täydellisesti. Tästä syystä konversiosarja kuuluu täydellisen vastaavuuden kategoriaan.

Konversiosarja muodostuu mahdollisesti seuraavista konversioista ja seuraavassa järjestyksessä**:**

```
Lvalue-muunnos ->
  Ylennys tai vakiokonversio ->
    Määreiden konversiot
```

Nimitys *lvalue-muunnos* viittaa kolmeen ensimmäiseen konversioon, jotka kuvattiin kohdassa 9.2 ja jotka kuuluvat täydellisen vastaavuuden kategoriaan: lvalue rvalueksi -muunnos, taulukko osoittimeksi -konversio ja funktio osoittimeksi -konversio. Konversiosarja on sellainen, jossa on yksi tai ei yhtään lvalue-muunnosta, jonka jälkeen tulee yksi tai ei yhtään ylennystä tai yksi vakiokonversio, jonka jälkeen tulee yksi tai ei yhtään määreen konversiota. Jokaisesta konversiosta käytetään enintään yhtä argumentin konvertoimiseksi vastaavan parametrin tyyppiseksi.

Tätä konversiosarjaa kutsutaan *vakiokonversiosarjaksi*. On olemassa myös toisenlainen konversiosarja, jota kutsutaan *käyttäjän määrittelemäksi konversiosarjaksi*. Käyttäjän määrittelemään konversiosarjaan kuuluu luokan konversiojäsenfunktio. Luokan konversiojäsenfunktiot ja käyttäjän määrittelemät konversiosarjat kuvataan luvussa 15.

Mitä ovat argumenttien konversiosarjat seuraavassa esimerkissä?

```
namespace libs_R_us {
    int max( int, int );
    double max( double, double );
}

// using-esittely
using libs_R_us::max;

void func()
{
    char c1, c2;
    max( c1, c2 ); // kutsuu funktiota libs_R_us::max( int, int )
}
```

Funktion max() kutsussa argumentit ovat char-tyyppisiä. Kun kutsutaan libs_R_us::max(int,int)-funktiota, on argumenttien konversiosarja seuraavanlainen:

1a. Koska argumentit välitetään arvoina, saadaan lvalue rvalueksi -konversiolla arvot argumenteista c1 ja c2.

2a. Ylennys konvertoi argumentit char-tyypistä int-tyypiksi.

Kun kutsutaan libs_R_us::max(double,double)-funktiota, on argumenttien konversiosarja seuraavanlainen:

1b. Lvalue rvalueksi -konversiolla saadaan arvot argumenteista c1 ja c2.

2b. Liukuluku-kokonais-vakiokonversiolla muunnetaan argumentit char-tyypistä double-tyypiksi.

Ensimmäisen konversiosarjan arvo on ylennys (sarjan huonompi konversio), kun taas toisen konversiosarjan arvo on vakiokonversio. Koska ylennys on parempi kuin vakiokonversio, valitaan libs_R_us::max(int,int)-funktio parhaiten elinkelpoiseksi funktioksi eli parhaiten kutsua vastaavaksi funktioksi.

Jos argumenttien konversiosarjojen paremmuusjärjestykseen laittamisesta ei ole apua yksilöitäessä yhtä elinkelpoista funktiota, joka vastaa argumenttien tyyppiä paremmin kuin muut elinkelpoiset funktiot, kutsu on moniselitteinen. Seuraavassa esimerkissä molemmat calc()-funktion ilmentymät vaativat seuraavan konversiosarjan:

1. Lvalue rvalueksi -konversio arvojen saamiseksi argumenteista i ja j.
2. Vakiokonversio argumenttien konvertoimiseksi vastaaviksi parametreiksi.

Koska kumpikin konversiosarja on yhtä hyvä kuin toinen, kutsu on moniselitteinen:

```
int i, j;
extern long calc( long, long );
extern double calc( double, double );

void jj() {
```

```
    // virhe: moniselitteinen, ei "parasta" vastaavuutta
    calc( i, j );
}
```

Määrekonversio — konversio, joka lisää const- tai volatile-määreen tyyppiin, johon osoitin osoittaa — arvo on täydellinen vastaavuus. Jos kuitenkin kaksi konversiosarjaa ovat identtisiä, paitsi että toisessa on lisämäärekonversio sarjan lopussa, silloin konversiosarja ilman lisämäärekonversiota on parempi. Esimerkiksi:

```
void reset( int * );
void reset( const int * );

int* pi;

int main() {
    reset( pi ); // sarja ilman määrekonversiota
                 // parempi: reset( int * ) valitaan
    return 0;
}
```

Vakiokonversiosarjan, jota käytetään ensimmäisen funktioehdokkaan, reset(int*), kutsun argumenttiin, katsotaan täsmäävän täydellisesti. Se vaatii ainoastaan lvalue rvalueksi -konversion, jotta arvo saadaan argumentista. Toiseen funktioehdokkaaseen, reset(const int *), käytetään myös lvalue rvalueksi -konversiota, jonka jälkeen tehdään määrekonversio, jotta osoitin int-tyyppiin saadaan osoittimeksi const int -tyyppiin. Nämä molemmat sarjat täsmäävät täydellisesti, mutta edellinen funktion kutsu on moniselitteinen. Koska molemmat konversiosarjat ovat identtisiä, paitsi että toisen konversiosarjan lopussa on lisämäärekonversio, pidetään sarjaa ilman määrekonversiota parempana. Täten reset(int*)-funktio on paras elinkelpoisista funktioista.

Tässä on toinen esimerkki, jossa määrekonversio vaikuttaa siihen, mikä konversiosarja valitaan:

```
int extract( void * );
int extract( const void * );

int* pi;

int main() {
    extract( pi ); // extract( void * ) tämä valitaan
    return 0;
}
```

Kutsuun on kaksi elinkelpoista funktiota: extract(void*) ja extract(const void*). Konversiosarja, jota käytetään ensimmäisen elinkelpoisen funktion, extract(void*):n, kutsuun, muodostuu lvalue rvalueksi -konversiosta, jolla saadaan arvo argumentista. Sen jälkeen tehdään vakio-osotinkonversio, jolla saadaan tämä arvo osoittimesta int-tyyppiin osoittimeksi void-tyyppiin. Konversiosarja, jota käytetään toisen elinkelpoisen funktion, extract(const void*), kutsuun, on identtinen, paitsi että siinä käytetään lisämäärekonversiota, joka konvertoi tuloksen osoit-

timesta void-tyyppiin osoittimeksi const void -tyyppiin. Koska nämä kaksi konversiosarjaa ovat identtisiä, paitsi että toisessa konversiosarjassa on lisämäärekonversio sarjan lopussa, valitaan ensimmäinen paremmaksi konversiosarjaksi. Funktio extract(void*) valitaan parhaiten elinkelpoiseksi funktioksi argumentille.

Määreet const ja volatile vaikuttavat myös siihen, kuinka viittausparametrin alustaminen sijoittuu paremmuusjärjestyksessä. Konversiosarjoissa asianlaita on niin, että jos kahden viittauksen alustukset ovat identtisiä, paitsi että toisessa on lisäksi const- tai volatile-määre, se viittauksen alustus, jossa ei ole lisämäärettä, on parempi funktion ylikuormituksen ratkaisun tarkoitukseen. Esimerkiksi:

```
#include <vector>
void manip( vector<int> & );
void manip( const vector<int> & );

vector<int> f();
extern vector<int> vec;

int main() {
    manip( vec ); // funktio manip( vector<int> & ) valitaan
    manip( f() ); // funktio manip( const vector<int> & ) valitaan
    return 0;
}
```

Viittauksen alustuksen ensimmäinen kutsu vastaisi täysin kumpaa tahansa funktiota. Mutta tämä kutsu ei ole moniselitteinen. Koska molemmat viittauksen alustukset ovat identtisiä, paitsi että toisessa on lisäksi const-määre, pidetään alustusta ilman lisämäärettä parempana alustuksena. Täten elinkelpoinen manip(vector<int>&)-funktio on paras funktio kutsua varten.

Toisessa kutsussa on olemassa vain yksi elinkelpoinen funktio: manip(const vector<int>&). Koska argumentti on tilapäinen, ja sisältää f()-funktion kutsun paluuarvon, argumentti on rvalue, jota ei voi käyttää manip(vector<int>&)-funktion const-määreettömän viittausparametrin alustamiseen. Parhaiten elinkelpoinen funktio toista kutsua varten on siten manip(const vector<int>&).

Funktion kutsuissa voi olla tietysti useampi kuin yksi argumentti. Parhaiten elinkelpoisen funktion valitsemiseksi pitää ottaa huomioon tarvittavien konversiosarjojen paremmuusjärjestys, joita tarvitaan kaikkien argumenttien konvertoimiseen. Tutkikaamme seuraavaa esimerkkiä:

```
extern int ff( char*, int );
extern int ff( int, int );

int main() {
    ff( 0, 'a' ); // ff( int, int )
    return 0;
}
```

Funktio ff(), joka saa kaksi int-tyyppistä argumenttia, valitaan parhaiten elinkelpoiseksi funktioksi seuraavien syiden takia:

1. Sen ensimmäinen argumentti on parempi. 0 vastaa täysin parametrin tyyppiä int, toisaalta se vaatii osoittimen vakiokonversiosarjan vastatakseen parametrin char*-tyyppiä.
2. Sen toinen argumentti on yhtä hyvä. Argumentti 'a' on char-tyyppinen ja vaatii konversiosarjan, jota arvostetaan samalla tavalla kuin ylennystä ja jolla se saadaan vastaamaan molempien funktioiden toista parametria.

Tässä on toinen esimerkki:

```
int compute( const int&, short );
int compute( int&, double );

extern int iobj;
int main() {
    compute( iobj, 'c' ); // compute( int&, double )
    return 0;
}
```

Nämä kaksi funktiota, compute(const int&, short) ja compute(int&, double), ovat elinkelpoisia funktioita. Toinen funktio valitaan parhaiten elinkelpoiseksi funktioksi seuraavista syistä:

1. Sen ensimmäinen argumentti on parempi. Viittauksen alustus ensimmäisen elinkelpoisen funktion kannalta on huonompi, koska siinä on const-määre eikä toisessa elinkelpoisessa funktiossa sitä ole.
2. Sen toinen parametri on yhtä hyvä. Argumentti 'c' on char-tyyppiä ja vaatii vakiokonversiosarjan, jotta se vastaisi molempien funktioiden toista parametria.

9.4.4 Oletusargumentit

Oletusargumentit voivat sallia enemmän funktioita elinkelpoisten funktioiden joukkoon. Elinkelpoinen funktio on sellainen, jota voidaan kutsua kutsussa määritetyllä argumenttiluettelolla. Elinkelpoisella funktiolla voi olla enemmän parametreja kuin funktion kutsussa olevassa argumenttiluettelossa on argumentteja, jos jokaiseen lisäparametriin liittyy oletusargumentti:

```
extern void ff( int );
extern void ff( long, int = 0 );

int main() {
    ff( 2L );   // vastaa funktiota ff( long, 0 );
    ff( 0, 0 ); // vastaa funktiota ff( long, int );
    ff( 0 );    // vastaa funktiota ff( int );
    ff( 3.14 ); // virhe: moniselitteinen
}
```

Vaikka argumenttiluettelossa on vain yksi argumentti, on ensimmäiselle ja kolmannelle kutsulle toinen funktio, ff(), elinkelpoinen funktio seuraavista syistä:

1. Funktion toiselle parametrille on olemassa oletusargumentti.
2. Sen ensimmäinen parametri on long-tyyppinen, mikä vastaa täsmälleen ensimmäisen kutsun argumentin tyyppiä ja kolmannen kutsun argumentin tyyppiä, kun sille tehdään vakiokonversion arvoinen konversiosarja.

Viimeinen kutsu on moniselitteinen, koska molemmat ilmentymät voivat täsmätä ensimmäisen argumentin vakiokonversion jälkeen. Funktio ff(int) ei ole parempi, koska sillä on täsmälleen yksi argumentti.

Harjoitus 9.9

Selitä, mitä tapahtuu funktion ylikuormituksen ratkaisun aikana, kun kutsutaan compute()-funktiota main()-funktiossa. Mitkä funktiot ovat funktioehdokkaita? Mitkä funktiot ovat elinkelpoisia funktioita? Minkä tyyppistä konversiosarjaa käytetään argumenttiin, jotta se vastaisi jokaisen elinkelpoisen funktion parametria? Mikä funktioista (jos yksikään) on parhaiten elinkelpoinen funktio?

```
namespace primerLib {
    void compute( );
    void compute( const void * );
}

using primerLib::compute;
void compute( int );
void compute( double, double = 3.4 );
void compute( char*, char* = 0 );

int main() {
    compute( 0 );
    return 0;
}
```

Mitä tapahtuisi, jos using-esittely sijaitsisi main()-funktiossa ennen kuin compute()-funktiota kutsutaan? Vastaa samoihin kysymyksiin, jotka kysyttiin aiemmin.