# Osa III

## *Proseduraalipohjainen ohjelmointi*

Osa II esitteli C++-ohjelmointikielen peruskomponentit: sisäiset tietotyypit (kuten int ja double), luokka-abstraktiotyypit (kuten string ja vector) ja operaatioita, joita voidaan käyttää näillä tyypeillä. Osassa III katsomme, kuinka näitä ohjelman komponentteja voidaan ryhmittää funktioiden määrittelyjen muodostamiseen. Funktioita käytetään, kun toteutetaan algoritmeja, jotka tekevät tiettyjä tehtäviä ohjelmissamme.

Jokaiseen C++-ohjelmaan pitää määritellä funktio nimeltään main(), joka on ensimmäinen käynnistettävä funktio, kun C++-ohjelma aloittaa suorituksensa. main() käynnistää muita funktioita, jotka tekevät ohjelmalta vaaditut tehtävät. Ohjelman funktiot kommunikoivat eli vaihtavat tietoja arvojen kautta, joita ne vastaanottavat (kutsutaan *parametreiksi*) ja palauttavat. Luvussa 7 esitellään C++:n funktiomekanismi.

Funktioita käytetään ohjelmien järjestämiseen pienemmiksi, itsenäisiksi yksiköiksi. Jokainen funktio kapseloi algoritmin tai joukon algoritmeja, joka käyttää tiettyä tietojoukkoa. Voimme esitellä olioita ja tyyppejä niin, että niitä voidaan käyttää koko ohjelman keston ajan. Mutta, jos näitä olioita ja tyyppejä käytetään vain osassa ohjelmaa, on parempi rajoittaa ne sinne, missä niitä käsitellään, ja liittää niiden esittelyt funktioihin, jotka niitä käyttävät. Viittausalue on mekanismi, jolla ohjelmoijat voivat rajoittaa ohjelmiensa esittelyiden näkyvyyttä. Luvussa 8 esittelemme C++:n tukemia eri viittausalueita; esittelemme myös, kuinka viittausalue vaikuttaa C++-olioiden esittelyihin, elinaikaan sekä suorituksenaikaisiin ominaisuuksiin.

C++:ssa on monia piirteitä, jotka helpottavat funktioiden käyttöä C++-ohjelmissa. Osassa III tarkastelemme näitä piirteitä jokaista vuorollaan. Ensimmäinen sellainen piirre on ylikuormitetut funktiot. Funktiot, jotka tekevät yhteisen operaation, mutta toimivat eri tietotyypeillä ja vaativat eri toteutuksen, voivat olla samannimisiä. Esimerkiksi funktiot, jotka tulostavat erityyppisiä arvoja kuten kokonaislukuja, merkkijonoja jne., voivat olla kaikki nimeltään print(). Tämä ominaisuus helpottaa funktioiden käyttöä, koska ohjelmoijien ei tarvitse muistaa eri funktioiden nimiä samalle operaatiolle. Kääntäjä valitsee sopivan kutsuttavan funktion käyttäen hyväkseen funktion argumentteja. Luvussa 9 tutkitaan, kuinka ylikuormitettuja funktioita esitellään ja käytetään sekä kuinka kääntäjä valitsee tietyn funktion ylikuormitettujen funktioiden joukosta.

Toinen C++:n tukema piirre funktioiden käytön helpottamiseksi on funktiomallit. Funktiomalli on geneerinen funktiomäärittely, jota voidaan käyttää automaattisesti loputtomasti sellaisten erilaisten funktioiden määrittelyihin, joiden tyypit voivat vaihdella, mutta joiden toteutus säilyy muuttumattomana. Luvussa 10 kuvataan, kuinka funktiomalleja määritellään ja käytetään funktiomäärittelyjen automaattiseen generointiin ja *instantiointiin*.

Ohjelman funktiot kommunikoivat ottamalla vastaan arvoja (eli parametreja) ja palauttamalla arvoja. Tämä mekanismi voi osoittautua kuitenkin riittämättömäksi, kun tapahtuu epätavallista eli ohjelma poikkeaa normaalista suorituksestaan ajon aikana. Sellaisessa tilanteessa, jota kutsutaan *poikkeukseksi* (*exception*), vaaditaan välittömiä toimenpiteitä ja sitä että funktio nopeasti kommunikoi sitä kutsuvalle funktiolle tapahtuneesta poikkeuksesta. C++:ssa on poikkeuksen käsittelypiirre, jolla voidaan kommunikoida funktioiden välillä näissä epätavallisissa tilanteissa. Poikkeusten käsittely esitellään luvussa 11.

Lopuksi, C++-vakiokirjastossa on mittava kokoelma usein käytettyjä funktioita, joita kutsutaan geneerisiksi algoritmeiksi. Luvussa 12 kuvataan näitä C++-vakiokirjaston geneerisiä algoritmeja ja tutkitaan, kuinka ne ovat vuorovaikutuksessa luvun 6 säiliötyyppien kuten myös sisäisen taulukkotyypin kanssa.

# 7

# *Funktiot*

Nyt, kun olemme nähneet, kuinka muuttujia esitellään (luku 3) ja kuinka lausekkeita (luku 4) ja lauseita (luku 5) kirjoitetaan, tässä luvussa katsotaan, kuinka näitä ohjelman komponentteja ryhmitetään funktiomäärittelyihin helpottamaan niiden uudelleenkäyttöä ohjelmissa. Tässä luvussa kuvataan, kuinka funktioita esitellään ja määritellään, kuten myös, kuinka niitä käynnistetään ohjelmissamme. Tämä luku esittelee erilaisia funktioparametreja, joita funktio voi ottaa vastaan ominaisuuksineen. Luvussa esitellään myös erityyppisiä arvoja, joita funktio voi palauttaa. Sitten tutkimme neljää erityistä funktiota: välittömät funktiot, rekursiiviset funktiot, linkitysdirektiiveissä esitellyt muut kuin C++-funktiot ja main()-funktio. Päätämme tämän luvun esittelemällä pitemmälle menevän aiheen: funktio-osoittimet.

## 7.1 Yleiskatsaus

Funktion voidaan ajatella olevan käyttäjän määrittelemä operaatio. Yleensä funktiota kuvataan nimellä. Funktion operandit, joita kutsutaan sen *parametreiksi*, määritetään sulkujen sisällä pilkuin eroteltuna *parametriluettelona*. Funktion tulosta kutsutaan *paluuarvoksi* ja paluuarvon tyyppiä kutsutaan *paluutyypiksi*. Funktiolla, joka ei johda mihinkään arvoon, on paluutyyppi void — se ei siis palauta mitään. Toimenpiteet, jotka funktio tekee, on määritetty funktion *runkoon*. Funktion runko on aaltosulkujen sisällä ja sitä kutsutaan joskus *funktiolohkoksi*. Funktion paluutyyppi, jonka jälkeen tulevat funktion nimi, parametriluettelo ja funktion runko, muodostavat *funktiomäärittelyn*. Tässä on joitakin esimerkkejä funktiomäärittelyistä:

```
inline int abs( int iobj )
{
    // palauta absoluuttinen arvo iobj
    return( iobj < 0 ? -iobj : iobj );
}

inline int min( int p1, int p2 )
{
    // palauta kahdesta arvosta pienempi
```

```
    return( p1 < p2 ? p1 : p2 );
}

int gcd( int v1, int v2 )
{
    // palauta suurin yhteinen jakaja
    while ( v2 )
    {
        int temp = v2;
        v2 = v1 % v2;
        v1 = temp;
    }
    return v1;
}
```

Funktiota kutsutaan aina, kun funktion nimen perässä on kutsuoperaattori ( () ). Jos funktio on määritelty vastaanottamaan parametreja, annetaan *argumentit* näihin parametreihin, kun funktiota kutsutaan. Nämä argumentit sijoitetaan kutsuoperaattorin sisään. Vierekkäiset argumentit erotetaan toisistaan pilkulla. Tätä järjestelyä kutsutaan *argumenttien välitykseksi* funktiolle. Seuraavassa esimerkissä main() kutsuu abs()-funktiota kahdesti ja funktioita min() ja gcd() kerran kumpaakin. Se on määritelty tiedostoon main.C.

```
#include <iostream>

int main()
{
    // pyydä arvot vakiosyötöstä
    cout << "Enter first value: ";
    int i;
    cin >> i;
    if ( !cin ) {
        cerr << "!? Oops: input error - Bailing out!\n";
        return -1;
    }

    cout << "Enter second value: ";
    int j;
    cin >> j;
    if ( !cin ) {
        cerr << "!? Oops: input error - Bailing out!\n";
        return -2;
    }

    cout << "\nmin: " << min( i, j ) << endl;
    i = abs( i );
    j = abs( j );
    cout << "gcd: " << gcd( i, j ) << endl;
    return 0;
}
```

Funktion kutsu voi saada aikaan jommankumman seuraavista: jos funktio on esitelty *välittömäksi* (*inline*), funktion runko on voitu laajentaa kutsukohtaan käännöksen aikana; ellei sitä ole esitelty välittömäksi, funktio käynnistetään suorituksen aikana. Funktion käynnistys saa aikaan ohjelman kontrollin siirtymisen käynnistettyyn funktioon; nykyisen aktiivisen funktion suoritus jää odottavaan tilaan. Kun kutsutun funktion suoritus on päättynyt, odottavassa tilassa ollut funktio jatkaa suoritustaan välittömästi kutsukohdan jälkeen. Funktion suoritus päättyy funktiorungon viimeisen lauseen suorituksen jälkeen tai kun kohdataan *return*-lause funktiorungossa.

Funktio pitää esitellä ohjelmalle, ennen kuin sitä kutsutaan; muussa tapauksessa tapahtuu käännöksenaikainen virhe. Funktion määrittely toimii tietysti sen esittelynä. Funktio voidaan kuitenkin määritellä ohjelmassa vain kerran. Määrittely sijaitsee tyypillisesti omassa ohjelmatekstitiedostossa, joka sisältää sen ja muiden siihen liittyvien funktioiden määrittelyt. Funktion esittelylle tarvitaan lisämekanismia avuksi, jos sitä käytetään muissa tiedostoissa kuin siinä, joka sisältää sen määrittelyn.

Funktion esittely muodostuu funktion paluutyypistä, funktion nimestä ja parametriluettelosta. Näitä kolmea elementtiä kutsutaan *funktion esittelyksi* eli *funktion prototyypiksi*. Funktio voidaan esitellä useita kertoja tiedostossa.

Jos main.C-esimerkkimme funktioita abs(), min() ja gcd() ei olisi määritelty ennen main()-funktiota, olisi jokainen niiden kutsu main()-funktiossa generoinut käännöksenaikaisen virheen. Jotta main.C kääntyisi virheittä, ei näiden funktioiden määrittelyä kuitenkaan vaadita ennen main()-funktiota; voisimme esitellä ne yksinkertaisesti seuraavasti (funktion esittelyyn ei tarvitse määrittää parametrien nimiä, vain jokaisen parametrin tyyppi):

```
int abs( int );
int min( int, int );
int gcd( int, int );
```

Funktioiden esittelyt (ja välittömien funktioiden määrittelyt) on parasta sijoittaa otsikkotiedostoihin. Näitä otsikkotiedostoja voidaan sitten ottaa mukaan jokaiseen tiedostoon, jossa näitä funktioita kutsutaan. Tällä tavalla kaikilla tiedostoilla on yhteinen esittely. Jos tuota esittelyä pitää muokata, vain tuota ilmentymää tarvitsee muuttaa. Ohjelmamme otsikkotiedosto voitaisiin määritellä kuten seuraavassa. Kutsukaamme sitä nimellä localMath.h:

```
// määrittely gcd.C:ssä
int gcd( int, int );

inline int abs(int i) {
    return( i<0 ? -i : i );
}

inline int min(int v1,int v2) {
    return( v1<v2 ? v1 : v2 );
}
```

Funktion esittely kuvaa funktion *rajapinnan* (*interface*). Se kuvaa ne erilaiset tiedot, joita funktion pitää vastaanottaa (parametriluettelo) ja antaa takaisin, jos antaa mitään (paluutyyppi). Funktion käyttäjinä me ohjelmoimme sen rajapinnan; koodiamme ei tarvitse muuttaa huolimatta siitä, kuinka usein funktion runkoa muokataan, edellyttäen, että funktion rajapinta säilyy ennallaan. Mekanismi, jolla kommunikoimme käyttäjien kanssa funktioiden rajapinnoilla, on funktioidemme esittelyiden sijoittaminen otsikkotiedostoihin kuten localMath.h.

Kun main.C-ohjelmamme käännetään ja suoritetaan, käyttäjän antamilla syöttöarvoilla

```
Enter first value:  15
Enter second value:  123
```

ohjelma tuottaa seuraavat tulokset:

```
min:  15
gcd:  3
```

## 7.2 Funktion prototyyppi

Funktion prototyyppi muodostuu funktion paluutyypistä, funktion nimestä ja parametriluettelosta. Funktion prototyyppi kuvaa funktion rajapinnan; se yksilöi parametrien lukumäärän ja tyypit, jotka pitää antaa funktiota kutsuttaessa sekä arvon tyypin, jonka funktio palauttaa. Tässä kohtaa tutkimme funktion prototyypin piirteitä yksityiskohtaisemmin.

### 7.2.1 Funktion paluutyyppi

Funktion paluutyyppi voi olla esimääritelty tyyppi, kuten int tai double, yhdistetty tyyppi, kuten int& tai double*, käyttäjän määrittelemä tyyppi, kuten lueteltu joukko, luokka tai void, joka tarkoittaa, että funktio ei palauta mitään tyyppiä. Seuraavassa on esimerkkejä mahdollisista funktion paluutyypeistä:

```
#include <string>
#include <vector>
class Date { /* määrittely */ };

bool look_up( int *, int );
double calc( double );
int count( const string &, char );
Date& calendar( const char* );
void sum( vector<int>&, int );
```

Paluutyypiksi ei voi määrittää funktiotyyppiä eikä sisäistä taulukkotyyppiä. Esimerkiksi seuraava on virheellinen:

```
// virhe: taulukko ei voi olla paluutyyppi
int[10] foo_bar();
```

Sen sijaan pitää palauttaa osoitin taulukon sisältämän elementin tyyppiin:

```
// ok: osoitin taulukon ensimmäiseen elementtiin
int *foo_bar();
```

Osoitin osoittaa palautettavan taulukon ensimmäiseen elementtiin. (Paluuarvon käsittely on käyttäjän vastuulla, jotta tiedettäisiin taulukon koko.)

Sen sijaan luokka- ja säiliötyyppejä voidaan palauttaa suoraan. Esimerkiksi:

```
// ok: paluutyyppi on lista merkkejä
list<char> foo_bar();
```

(Tämä lähestymistapa on kuitenkin tehotonta. Katso käsittelymme arvon palautuksesta kohdasta 7.4.)

Funktion pitää määrittää paluuarvo. Esittely tai määrittely ilman eksplisiittistä paluuarvoa johtaa käännöksenaikaiseen virheeseen. Esimerkiksi:

```
// virhe: paluutyyppi puuttuu
const is_equal( vector<int> v1, vector<int> v2 );
```

C++-esistandardissa oletettiin, että jos paluutyyppi puuttui, se oli int. C++-standardissa funktion tyyppiä ei voi jättää pois. Funktion is_equal() oikea esittely on

```
// ok: paluutyyppi on määritetty
const bool is_equal( vector<int> v1, vector<int> v2 );
```

### 7.2.2 Funktion parametriluettelo

Funktion parametriluetteloa ei voi jättää pois. Funktio, jolla ei ole yhtään parametria, voidaan esittää joko tyhjällä parametriluettelolla tai yhdellä void-avainsanalla. Esimerkiksi seuraavat fork()-funktion esittelyt ovat samanarvoisia:

```
int fork();      // implisiittinen void-parametriluettelo
int fork( void ); // samanarvoinen esittely
```

Parametriluettelo muodostuu pilkuin erotelluista parametrityypeistä. Jokaisen parametrityypin jälkeen voidaan antaa valinnainen nimi. Parametriluettelon lyhennetty syntaksi on virheellinen. Esimerkiksi:

```
int manip( int v1, v2 );    // virhe
int manip( int v1, int v2 ); // ok
```

Parametriluettelossa ei voi olla samannimisiä parametreja. Parametrin nimi funktion määrittelyn parametriluettelossa mahdollistaa parametrin käsittelyn funktion rungossa. Parametrin nimi on tarpeeton funktion esittelyssä. Jos se on annettu, sen tulisi toimia dokumentaation apuna. Esimerkiksi:

```
void print( int *array, int size );
```

Kieli ei rankaise siitä, jos määritetään parametrin nimi eri tavalla saman funktion esittelyssä ja määrittelyssä. Ohjelman lukija voi kuitenkin hämmentyä.

C++:ssa on mahdollista, että kahdella funktiolla on sama nimi, mutta erilaiset parametriluettelot. Tällöin funktioita kutsutaan *ylikuormitetuiksi funktioiksi* (*overloaded functions*). Parametriluetteloa kutsutaan funktion *tunnisteeksi* (*signature*), koska se erottaa funktion ilmentymän muista. Funktion nimi ja tunniste yksilöivät sen. Luvussa 9 käsitellään ylikuormitettuja funktioita täydellisemmin.

### 7.2.3 Parametrien tyyppitarkistus

Funktio gcd() on esitelty seuraavasti:

```
int gcd( int, int );
```

Esittely ilmaisee, että funktiolla on kaksi int-tyyppistä parametria. Funktion parametriluettelossa on tarpeellinen määrä tietoa kääntäjälle funktiolle annettujen argumenttien tyyppitarkistuksen tekemiseen, kun funktiota kutsutaan.

Mitä esimerkiksi tapahtuu, jos argumentit ovat tyyppiä const char*? Mitä voisi olla tuloksena seuraavasta kutsusta?

```
gcd( "hello", "world" );
```

Tai mitä tapahtuu, jos funktiolle gcd() välitetään vain yksi tai useampi kuin kaksi argumenttia? Mitä tapahtuu, jos vahingossa yhdistämme tämän kutsun kaksi arvoa, 24 ja 312?

```
gcd( 24312 );
```

Ainoa toivottava tulos yrityksestä kääntää nämä kaksi viimeistä gcd()-kutsua on käännöksenaikainen virhe; kaikki yritykset suorittaa nämä kutsut johtavat tuhoon. C++:ssa nämä kaksi kutsua johtavat käännöksenaikaiseen virheeseen, joiden ilmoitukset ovat seuraavaa yleistä muotoa:

```
// gcd( "hello", "world" )
error: invalid argument types ( const char*, const char* ) --
     expecting ( int, int )

// gcd( 24312 )
error: missing value for second argument
```

Mitä tapahtuu, jos argumentit ovat double-tyyppisiä? Tulisiko kutsun aiheuttaa virheilmoitus?

```
gcd( 3.14, 6.29 );
```

Kuten näimme kohdassa 4.14, double-tyypin arvo voidaan konvertoida int-tyyppiseksi. Siitä syystä virheilmoituksen generoiminen kutsusta on liian vakava. Sen sijaan argumentit konvertoidaan implisiittisesti int-tyyppisiksi (katkaisun kautta), mikä täyttää parametriluettelon tyyppivaatimukset. Koska kuitenkin kyseessä on arvoa pienentävä konversio, jossa tapahtuu mahdollisesti tarkkuuden menetys, kääntäjä antaa yleensä varoituksen. Kutsusta tulee

```
gcd( 3, 6 );
```

ja se palauttaa arvon 3.

C++ on *vahvasti tyypitetty* kieli. Jokaisen funktion argumentit *tyyppitarkistetaan* käännöksen aikana. Jos tyypit eivät vastaa toisiaan (*type mismatch*) argumentin ja vastaavan funktion parametrin välillä, tehdään implisiittinen konversio, jos se on mahdollista, kuten edellisessä esimerkissä double-tyyppi konvertoitiin int-tyypiksi. Jos implisiittinen konversio ei ole mahdollinen tai jos argumenttien lukumäärä on väärä, annetaan käännöksenaikainen virheilmoitus. Tästä syystä funktiota ei voi kutsua, ennen kuin se on ensiksi esitelty. Esittely on tarpeellinen kääntäjälle funktiokutsun argumenttien tyyppien tarkistamiseen funktion parametriluetteloa vastaan.

Argumentin pois jättäminen tai vääräntyyppisen argumentin välittäminen ovat yleisiä vakavien ajonaikaisten ohjelmavirheiden lähteitä esistandardin mukaisessa C-kielessä. C++:n esittelemän vahvan tyyppitarkistuksen avulla nämä rajapintavirheet saadaan kiinni käännöksen aikana.

### Harjoitus 7.1

Mitkä seuraavista ovat virheellisiä funktion prototyyppejä, vai onko yksikään? Miksi?

```
(a) set( int *, int );
(b) void func();
(c) string error( int );
(d) arr[10] sum( int *, int );
```

### Harjoitus 7.2

Kirjoita prototyypit jokaiselle seuraavalle funktiolle:

(a) Funktio on nimeltään compare, ja sillä on kaksi parametria. Parametrit ovat viittauksia luokkaan nimeltään matrix ja paluutyyppi on bool-tyyppinen.

(b) Funktio on nimeltään extract, sillä ei ole parametreja ja se palauttaa joukon kokonaislukuja (jossa joukko, set, on kohdassa 6.13 määritelty säiliötyyppi).

### Harjoitus 7.3

Kun on annettu seuraavat esittelyt, mitkä funktioiden kutsuista ovat virheellisiä, vai onko yksikään? Miksi?

```
double calc( double );
int count( const string &, char );
void sum( vector<int> &, int );
vector<int> vec( 10 );

(a) calc( 23.4, 55.1 );
(b) count( "abcda", 'a' );
(c) sum( vec, 43.8 );
(d) calc( 66 );
```

## 7.3 Argumenttien välitys

Funktiot käyttävät varattua muistia ohjelman *suorituksenaikaisesta pinosta.* Tuo muistialue säilyy funktioon liitettynä, kunnes funktio päättyy. Siinä vaiheessa muistitila annetaan uudelleen käytettäväksi. Funktion koko muistialuetta kutsutaan *aktivointitietueeksi* (*activation record*).

Jokaiselle funktion parametrille annetaan muistia aktivointitietueelta. Parametrin muistialueen koko riippuu sen tyypistä. Argumenttien välitys on funktion parametrien muistialueen alustamista funktion kutsun argumenttien arvoilla.

Argumenttien välityksen alustuksen oletusmetodi C++:ssa on argumenttien kopiointi parametrien muistialueelle. Tätä kutsutaan *arvovälitykseksi* (*pass-by-value*).

Kun argumentit välitetään arvoina, ei funktio koskaan käsittele kutsun argumentteja. Arvot, joita funktio käsittelee, ovat sen omia, paikallisia kopioita ja ne ovat tallennettuina suorituksenaikaiseen pinoon. Näihin arvoihin tapahtuvat muutokset eivät vaikuta argumenttien arvoihin. Kun funktio päättyy ja funktion aktivointitietue vedetään pois ajonaikaisesta pinosta, nämä paikalliset arvot katoavat.

Kun argumentit välitetään arvoina, argumenttien sisällöt eivät muutu. Tämä tarkoittaa, että ohjelmoijan ei tarvitse tallentaa ja palauttaa argumentteja tehdessään funktion kutsun. Ilman arvovälitysmekanismia jokainen parametri, jota ei ole esitelty määreellä const, voi joutua mahdollisesti muutetuksi jokaisessa funktion kutsussa. Arvovälityksestä aiheutuu vähiten mahdollista harmia ja se vaatii vähiten työtä tavalliselta käyttäjältä. Arvovälitys on järkevä oletusmekanismi argumenttien välitykseen.

Arvovälitys ei kuitenkaan ole sopiva jokaiseen tilanteeseen. Tilanteisiin, joissa arvovälitys ei ole sopiva ratkaisu, kuuluvat seuraavat:

- Kun suuri luokkaolio pitää välittää argumenttina. Hinta, joka aiheutuu ajasta ja muistitilasta, kun luokkaoliota kopioidaan pinoon, on usein liian korkea tosielämän sovelluksiin.
- Kun argumenttien arvoja pitää muokata. Esimerkiksi swap()-funktiolla käyttäjä haluaa vaihtaa argumenttien arvot, mutta ei voi tehdä niin arvovälityksellä.

```
// swap() ei vaihda argumenttien arvoja keskenään!
void swap( int v1, int v2 ) {
    int tmp = v2;
    v2 = v1;
    v1 = tmp;
}
```

swap() vaihtaa keskenään argumenttien paikalliset kopiot. Muuttujat, jotka edustavat swap()-funktion argumentteja, säilyvät muuttumattomina. Tätä kuvaa seuraava ohjelma, joka kutsuu swap()-funktiota:

```
#include <iostream>
void swap( int, int );
```

```
int main() {
    int i = 10;
    int j = 20;

    cout << "Before swap():\ti: "
        << i << "\tj: " << j << endl;

    swap( i, j );

    cout << "After swap():\ti: "
        << i << "\tj: " << j << endl;
    return 0;
}
```

Kun tämä ohjelma käännetään ja suoritetaan, ohjelma tuottaa seuraavan tulostuksen:

```
Before swap():    i: 10    j: 20
After swap():     i: 10    j: 20
```

Jotta saavutettaisiin haluttu toiminta, ohjelmoijalla on käytettävissään kaksi vaihtoehtoa. Toisessa niistä parametrit esitellään osoittimina. swap() voitaisiin kirjoittaa uudelleen esimerkiksi näin:

```
// pswap() vaihtaa keskenään arvot, joita v1 ja v2 osoittavat
void pswap( int *v1, int *v2 ) {
    int tmp = *v2;
    *v2 = *v1;
    *v1 = tmp;
}
```

main()-funktiota pitää muokata niin, että se kutsuu pswap()-funktiota. Ohjelmoijan pitää nyt välittää kahden olion osoitteet eikä itse olioita:

```
pswap( &i, &j );
```

Kun uudistettu ohjelma käännetään ja suoritetaan, tulostus näyttää oikealta:

```
// osoitinten käyttö mahdollistaa, että ohjelmoija voi
// käsitellä kutsun argumentteja

Before swap():    i: 10    j: 20
After swap():     i: 20    j: 10
```

Toinen vaihtoehto on esitellä parametrit viittauksina. swap() voitaisiin kirjoittaa uudelleen esimerkiksi näin:

```
// rswap() vaihtaa keskenään arvot, joihin v1 ja v2 osoittavat
void rswap( int &v1, int &v2 ) {
    int tmp = v2;
    v2 = v1;
    v1 = tmp;
}
```

rswap()-funktion kutsu main()-funktiossa näyttää alkuperäisen swap()-funktion kutsulta:

```
rswap( i, j );
```

Kun ohjelma käännetään ja suoritetaan, ohjelma näyttää, että arvot ovat vaihtuneet keskenään oikein.

### 7.3.1 Viittausparametrit

Parametrin esittely viittauksena (*reference*) korvaa argumenttien oletusarvoisen arvovälitysmekanismin. Arvovälityksessä funktio käsittelee argumenttien paikallisia kopioita. Kun parametrit ovat viittauksia, funktio vastaanottaa argumentin *lvalue*-arvon kopion sijasta. Tämä tarkoittaa, että funktio tietää, missä argumentti sijaitsee muistissa ja voi siten muuttaa sen arvoa tai ottaa sen osoitteen.

Milloin on sopivaa määrittää parametri viittauksena? Se on sopivaa tapauksissa kuten swap(), jolloin muussa tapauksessa olisi tarpeellista vaihtaa parametrit osoittimiksi, jotta argumenttien arvojen muutokset olisivat mahdollisia. Toinen yleinen viittausparametrien käyttötilanne on lisätulosten palautus kutsuvaan funktioon. Kolmas käyttötilanne on suurten luokkaolioiden välitys funktioon. Tutkikaamme näitä kahta viimeistä tilannetta tarkemmin.

Esimerkkinä funktiosta, joka käyttää viittausparametria lisätuloksen palauttamiseen kutsuvaan funktioon, määritelkäämme funktio nimeltään look_up(), joka etsii tiettyä arvoa kokonaislukuvektorista. Jos arvo löytyy, look_up() palauttaa iteraattorin, joka viittaa arvon sisältävän vektorin elementtiin; muussa tapauksessa se palauttaa iteraattorin, joka viittaa yhden yli vektorin viimeisen elementin ilmaistakseen, että arvoa ei löytynyt. Jos esiintymiä on useampia, palautetaan iteraattori, joka osoittaa ensimmäiseen niistä. Lisäksi look_up() palauttaa viittausparametrin occurs, joka ilmaisee esiintymien lukumäärän.

```
#include <vector>

// viitausparametri, 'occurs', saattaa
// sisältää toisen paluuarvon

vector<int>::const_iterator look_up(
    const vector<int> &vec,
    int value,    // onko arvo vektorissa?
    int &occurs ) // kuinka monta kertaa?
{
    // res_iter alustetaan yhden yli viimeisen elementin
    vector<int>::const_iterator res_iter = vec.end();
    occurs = 0;

    for ( vector<int>::const_iterator iter = vec.begin();
        iter != vec.end();
        ++iter )
        if ( *iter == value )
        {
            if ( res_iter == vec.end() )
```

```
                res_iter = iter;
            ++occurs;
        }

    return res_iter;
}
```

Kolmas tilanne, jossa parametrin esittely viittauksena on järkevää, on silloin, kun välitämme suuria luokkaolioita funktiolle. Kun käytetään arvovälitystä, koko olio kopioidaan jokaisessa kutsussa. Vaikka arvovälitys on tyydyttävä ratkaisu sisäisten tyyppien ja pienten luokkien olioille, se on liian tehotonta suurille luokkaolioille. Viittausparametrin avulla funktiolla on pääsy argumenttina määritettyyn olioon eikä paikallisia kopioita tehdä funktion aktivointitietueelle. Esimerkiksi:

```
class Huge { public: double stuff[1000]; };
extern int calc( const Huge & );

int main() {
    Huge table[ 1000 ];
    // ... alusta taulukko

    int sum = 0;
    for ( int ix=0; ix < 1000; ++ix )
        // funktio calc() viittaa Huge-tyyppisen taulukon
        // elementtiin, joka on määritetty argumenttina
        sum += calc( table[ix] );

    // ...
}
```

Joku voisi haluta käyttää viittausparametria estääkseen argumenttina käytetyn suuren luokkaolion kopioinnin ja samaan aikaan haluaisi estää funktiota muuttamasta argumenttia. Aina, kun ei ole aikomusta muuttaa viittausparametria kutsutussa funktiossa, on hyvä käytäntö esitellä viittausparametri const-tyyppiseksi. Tämä menettely mahdollistaa sen, että kääntäjä voi estää tahattomien muutosten tekemisen. Esimerkiksi seuraava koodikatkelma rikkoo foo()-funktion xx-parametrin const-tyyppisyyttä. Koska foo_bar()-funktion parametri ei ole viittaus const-tyyppiin, ei ole takuita siitä, ettei foo_bar() muuttaisi argumenttinsa xx arvoa. Tämä rikkoo foo()-funktion xx-parametrin const-tyyppisyyttä ja kääntäjä antaa siitä virheilmoituksen:

```
class X;
extern int foo_bar( X& );

int foo( const X& xx ) {
    // virhe: const välitetään const:ittomalle
    return foo_bar( xx );
}
```

Jotta tämä ohjelma kääntyisi, voimme muuttaa foo_bar()-funktion parametrin tyyppiä; kumpikin seuraavista esittelyistä ovat hyväksyttäviä:

```
extern int foo_bar( const X& );
extern int foo_bar( X ); // arvovälitys
```

Tai voimme välittää argumentin, joka on xx:n kopio, jota foo_bar()-funktion sallitaan muuttaa:

```
int foo( const X &xx ) {
    // ...
    X x2 = xx; // kopioi arvot

    // kun foo_bar() muuttaa viittausparametriaan,
    // x2 muokkautuu; xx säilyy muuttumattomana
    return foo_bar( x2 ); // ok
}
```

Viittausparametri voidaan esitellä mihin tahansa sisäiseen tietotyyppiin. On esimerkiksi mahdollista esitellä parametri, joka on viittaus osoittimeen; tällöin ohjelmoija haluaisi muokata itse osoitinta sen osoittaman olion sijasta. Tässä on esimerkiksi funktio, joka vaihtaa kaksi osoitinta keskenään:

```
void ptrswap( int *&v1, int *&v2 ) {
    int *tmp = v2;
    v2 = v1;
    v1 = tmp;
}
```

Esittely

```
int *&v1;
```

tulisi lukea oikealta vasemmalle: v1 on viittaus osoittimeen, joka on osoitin olioon, joka on tyyppiä int. Kun funktiota main() käyttäen käsittelemme rswap()-funktiota, voimme muokata sen toteutusta niin, että se vaihtaa osoitinten arvot keskenään seuraavasti:

```
#include <iostream>
void ptrswap( int *&v1, int *&v2 );

int main() {
    int i = 10;
    int j = 20;

    int *pi = &i;
    int *pj = &j;

    cout << "Before ptrswap():\tpi: "
        << *pi << "\tpj: " << *pj << endl;

    ptrswap( pi, pj );
    cout << "After ptrswap():\tpi: "
        << *pi << "\tpj: " << *pj << endl;
```

```
    return 0;
}
```

Kun ohjelma käännetään ja suoritetaan, se generoi seuraavan tulostuksen:

```
Before ptrswap():    pi: 10    pj: 20
After ptrswap():     pi: 20    pj: 10
```

### 7.3.2 Viittaus- ja osoitinparametrien välinen suhde

Nyt saatat ihmetellä, kuinka päätät, esitteletkö funktion parametrin viittauksena vai osoittimena. Loppujen lopuksi molemmat parametritavat sallivat funktion muokata olioita, joihin argumentit viittaavat. Ja molemmat parametritavat mahdollistavat suurten luokkaolioiden välityksen funktioon tehokkaasti. Joten, kuinka päätämme, esittelemmekö funktion parametrin viittauksena vai osoittimena?

Kuten mainittiin kohdassa 3.6, pitää viittaus alustaa viittaamaan olioon, eikä sitä sen jälkeen voi koskaan asettaa viittaamaan muuhun olioon. Osoitin voi osoittaa peräkkäin eri olioita tai ei oliota ollenkaan.

Koska osoitin voi osoittaa joko olioon tai ei olioon ollenkaan, ei funktio voi turvallisesti käyttää osoitinta käänteisesti ennen kuin se ensiksi varmistuu siitä, että osoitin todella osoittaa olioon. Esimerkiksi:

```
class X;
void manip( X *px )
{
    // varmistu, että osoittimen arvo on nollasta poikkeava ennen käänteistä osoitusta
    if ( px != 0 )
        // käytä osoitinta käänteisesti
}
```

Toisaalta viittausparametrin ollessa kyseessä funktion ei tarvitse vartioida sitä, viittaako se olioon. Viittauksen pitää kohdistua olioon, vaikka muuta haluaisimmekin. Esimerkiksi:

```
class Type { };
void operate( const Type& p1, const Type& p2 );

int main() {
    Type obj1;
    // aseta obj1:een jokin arvo

    // virhe: viittausparametrin argumentti ei voi olla 0
    Type obj2 = operate( obj1, 0 );
}
```

Jos parametri voi viitata funktiossa eri olioihin tai jos parametrin ei tarvitse viitata mihinkään olioon, pitää käyttää osoitinparametria.

Eräs tärkeä viittausparametrien käyttötapa on ylikuormitettujen operaattorien tehokas toteutus, jolla yritetään samalla pitää operaattoreiden käyttöä helposti ymmärrettävinä. Ylikuormitettujen operaattorien täydellinen käsittely on luvussa 15. Päästäksemme alkuun, tutkikaamme seuraavaa esimerkkiä, jossa käytetään Matrix-luokkatyyppiä. Haluaisimme tukea kahden Matrix-olion sekä lisäys- että sijoitusoperaattoreita ja sallia niiden käytön yhtä "luonnollisina" kuin sisäisillä tyypeillä:

```
Matrix a, b, c;

c = a + b;
```

Matrix-luokkatyypin lisäys- ja sijoitusoperaatiot on toteutettu ylikuormitettuja operaattoreita käyttäen. Ylikuormitettu operaattori on funktio, jolla on hassu nimi. Lisäysoperaattorimme kohdalla funktion nimi on operator+. Tehkäämme määrittely tälle ylikuormitetulle operaattorille:

```
Matrix              // lisäys palauttaa Matrix-olion
operator+(   // ylikuormitetun operaattorin nimi
  Matrix m1, // vasemmanpuoleisen operandin tyyppi
  Matrix m2 // oikeanpuoleisen operandin tyyppi
)
{
     Matrix result;
     // laske tulos (result)
     return result;
}
```

Tämä toteutus tukee kahden Matrix-olion yhteenlaskua kuten tässä

```
a + b;
```

mutta valitettavasti se on anteeksiantamattoman tehoton. Huomaa, että operaattorimme parametrit eivät ole viittauksia; tämä tarkoittaa, että operaattorimme argumentit annetaan arvovälityksenä. Kahden Matrix-olion, a ja b, sisällöt kopioidaan operator+()-funktion parametrialueelle. Koska Matrix-luokka on melko suuri, aikaa ja muistia kuluu sen kopioimiseen ja muistin varaamiseen funktion parametrialueelta, eikä sitä siksi voida pitää hyväksyttävänä metodina.

Uskotellaan, että parantaaksemme operaattorifunktiomme tehokkuutta, päätämme esitellä parametrit osoittimina. Tältä näyttää uusi toteutuksemme operator+()-funktiosta:

```
// uusi toteutus osoitinparametreilla
Matrix operator+( Matrix *m1, Matrix *m2 )
{
     Matrix result;
     // laske tulos (result)
     return result;
}
```

Tässä toteutuksessa on kuitenkin seuraava ongelma: vaikka olemme saavuttaneet tehokkuutta, se on tapahtunut lisäysoperaattorin helpon käytön kustannuksella. Osoitinparametrit vaativat nyt, että välitämme yhteenlaskettavien Matrix-olioiden argumentit osoitteina. Yhteenlaskuoperaatiomme pitää nyt ohjelmoida näin:

```
&a + &b; // ei hyvä, vaan myös mahdoton
```

Vaikka tämä on kehnon näköistä ja johtaa todennäköisesti joidenkin ohjelmoijien valitukseen sen käyttäjäystävällisyyden puuttumisesta, yrityksestä lisätä kolme oliota yhdistetyssä lausekkeessa tulee erittäin vaikeaa:

```
// hups: tämä ei toimi
// yhteenlaskun &a + &b tyyppi on Matrix-olio
&a + &b + &c;
```

Saadakseen kolmen olion yhteenlaskun toimimaan hyvin osoitinratkaisussa, pitää ohjelmoijan kirjoittaa seuraavasti:

```
// ok: tämä toimii, mutta ...
&( &a + &b ) + &c;
```

Tietystikään ei voi odottaa kenenkään kirjoittavan noin. Viittausparametrit ovat tarvitsemamme ratkaisu. Kun parametri on viittaus, funktio vastaanottaa argumentin *lvaluen* sen kopion sijasta. Koska funktio tietää, missä kohdassa muistia argumentti sijaitsee, ei argumentin arvoa kopioida funktion parametrialueelle. Viittausparametrin argumentti on itse Matrix-olio; tämä sallii meidän käyttää lisäysoperaattoria luonnollisella tavalla, kuten käytämme sitä sisäisten tyyppien olioille.

Tässä on uudistettu luonnoksemme Matrix-luokan ylikuormitetusta lisäysoperaattorista:

```
// uusi toteutus viittausparametreilla
Matrix operator+( const Matrix &m1, const Matrix &m2 )
{
    Matrix result;
    // tee laskenta (result)
    return result;
}
```

Tämä toteutus tukee Matrix-olioiden yhteenlaskua kuten seuraavassa:

```
a + b + c;
```

Viittaukset esiteltiin C++:ssa eksplisiittisesti luokkatyyppien tueksi — erityisesti tukemaan selkeää, mutta tehokasta ylikuormitettujen operaattoreiden toteutusta.

### 7.3.3 Taulukkoparametrit

C++:ssa ei taulukoita koskaan välitetä arvoina. Sen sijaan taulukko välitetään osoittimena sen ensimmäiseen — nollanteen — elementtiin. Esimerkiksi tätä

```
void putValues( int[ 10 ] );
```

kääntäjä käsittelee, kuin se olisi esitelty näin

```
void putValues( int* );
```

Taulukon koko ei kuulu parametrin esittelyyn. Siten seuraavat kolme esittelyä ovat samanarvoisia:

```
// kolme samanarvoista esittelyä putValues()-funktiosta
void putValues( int* );
void putValues( int[] );
void putValues( int[ 10 ] );
```

Koska taulukko välitetään osoittimena, sillä on kaksi seuraamusta ohjelmoijille:

- Taulukkoparametriin tehdyt muutokset kutsutussa funktiossa tapahtuvat itse taulukkoargumenttiin eikä paikalliseen kopioon. Kun argumenttina olevan taulukon pitää pysyä muuttumattomana, ohjelmoijien pitää huolehtia taulukon alkuperäisestä kopiosta. Vaihtoehtoisesti funktio voisi ilmaista, että se ei aio muuttaa taulukon elementtejä ja esittelee parametrin elementit const-tyyppisiksi:

  ```
  void putValues( const int[ 10 ] );
  ```

- Taulukon koko ei ole osa parametrin tyyppiä. Funktio, jolle taulukko välitetään, ei tiedä sen todellista kokoa, eikä tiedä kääntäjäkään. Kun kääntäjä käyttää parametrin tyyppitarkistusta argumentin tyypille, ei taulukoiden kokoja tarkisteta. Esimerkiksi:

  ```
  void putValues( int[ 10 ] ); // kohdellaan kuten int*
  int main() {
      int i, j[ 2 ];
      putValues( &i ); // ok: &i on int*; potentiaalinen ajonaikainen virhe
      putValues( j );  // ok: j konvertoidaan osoittimeksi 0:teen
                       //     elementtiin; argumentin tyyppi on int*;
                       //     potentiaalinen ajonaikainen virhe
      return 0;
  }
  ```

Parametrien tyyppitarkistus vahvistaa, että molempien putValues()-funktion kutsujen argumentit ovat int*-tyyppisiä. Tyyppitarkistus ei totea, että argumentti on kymmenen elementin taulukko.

Sovittu on, että C-tyyliset merkkijonot ovat merkkitaulukoita, jotka koodataan päättyvän null-merkillä. Kuitenkin kaikkien muiden taulukkotyyppien mukaan lukien merkkitaulukot, joita halutaan käsitellä upotettuine null-merkkeineen, pitää jollain tavalla ilmaista kokonsa, kun niitä välitetään argumentteina funktioille. Eräs yleinen mekanismi on antaa lisäargumentti, joka sisältää taulukon koon. Esimerkiksi:

```
void putValues( int[], int size );
int main() {
    int i, j[ 2 ];
    putValues( &i, 1 );
    putValues( j, 2 );
    return 0;
}
```

putValues() tulostaa taulukon arvot seuraavassa muodossa:

```
( 10 )< 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 >
```

jossa 10 edustaa taulukon kokoa. Tässä toteutuksessa käytetään lisäparametria taulukon kokoa varten:

```
#include <iostream>

const lineLength = 12; // elementit riville
void putValues( int *ia, int sz )
{
    cout << "( " << sz << " )< ";
    for ( int i = 0; i < sz; ++i )
    {
        if ( i % lineLength == 0 && i )
            cout << "\n\t"; // rivi täysi

        cout << ia[ i ];

        // erota muut paitsi viimeinen elementti
        if ( i % lineLength != lineLength-1 &&
            i != sz-1 )
            cout << ", ";
    }
    cout << " >\n";
}
```

Toinen mekanismi on parametrin esittely viittauksena taulukkoon. Kun parametri on viittaus taulukkotyyppiin, taulukon koosta tulee osa parametria ja parametrityyppiä, jolloin kääntäjä tarkistaa, että taulukkoargumentin koko täsmää vastaavan funktion parametrityypin kanssa.

```
// parametri on viittaus taulukkoon, jossa on 10 kokonaislukua
void putValues( int (&arr)[10] );
int main() {
    int i, j[ 2 ];
    putValues( i ); // virhe: argumentti 10 kokonaisluvun taulukko
    putValues( j ); // virhe: argumentti 10 kokonaisluvun taulukko
    return 0;
}
```

Koska taulukon koko on nyt osa parametrin tyyppiä, hyväksyy tämä putValues()-versio vain taulukot, joissa on kymmenen int-kokonaislukua. Tämä rajoittaa taulukoita, joita voidaan välittää argumentteina putValues()-funktiolle. Se kuitenkin mahdollistaa merkittävästi yksinkertaisemman toteutuksen:

```
#include <iostream>

void putValues( int (&ia)[10] )
{
    cout << "( 10 )< ";
    for ( int i = 0; i < 10; ++i ) {
        cout << ia[ i ];

        // erota kaikki muut paitsi viimeinen elementti
        if ( i != 9 )
            cout << ", ";
    }
    cout << " >\n";
}
```

Vielä eräs mekanismi on abstraktin säiliötyypin käyttö. (Abstraktit säiliötyypit on esitelty luvussa 6.) Tätä mekanismia tutkitaan enemmän seuraavassa alikohdassa.

Vaikka kaksi edellistä putValues()-funktion toteutusta toimivat, on niillä joitakin vakavia rajoituksia. Ensimmäinen toteutus toimii vain taulukoilla, jotka ovat int-tyyppisiä. Tarvitaan toinen funktio, joka käsittelee double-taulukoita ja kolmas long-taulukoita jne. Toinen toteutus toimii vain taulukoille, joissa on kymmenen int-kokonaislukua. Jälleen tarvitaan lisäfunktioita, joilla käsiteltäisiin erikokoisia taulukoita. Parempi toteutus on määritellä putValues() funktiomalliksi. Funktiomalli on sellainen, jonka koodi säilyy muuttumattomana erilaisista parametrityypeistä huolimatta. Seuraavassa näytetään, kuinka ensimmäinen putValues()-versiomme voitaisiin kirjoittaa uudelleen funktiomallina, joka pystyy käsittelemään erityyppisiä ja -kokoisia taulukoita:

```
template <class Type>
    void putValues( Type *ia, int sz )
```

```
{
    // kuten ennenkin
}
```

Malliparametri tulee kulmasulkujen sisään; tässä tapauksessa malliparametri on Type. Avainsana class ilmaisee, että malliparametri edustaa tyyppiä. Type-tunnus toimii parametrin nimenä ja sen esiintymä putValues()-funktion parametriluettelossa toimii paikanpitäjänä todellisille tyypeille, kun funktiomalli *instantioidaan* (muodostetaan ilmentymä). Jokaisessa instantioinnissa Type-parametri korvataan todellisella tyypillä — int, double, string jne. Katsomme lisää funktiomalleja luvussa 10.

Parametri voi olla myös moniulotteinen taulukko. Sellaisen parametrin pitää määrittää kaikkien ulottuvuuksiensa koot paitsi ensimmäisen. Esimerkiksi

```
void putValues( int matrix[][10], int rowSize );
```

esittelee matrix:in kaksiulotteiseksi taulukoksi. Jokainen rivi muodostuu kymmenestä sarakeelementistä. matrix voidaan esitellä yhtäpitävästi näin:

```
int (*matrix)[10]
```

Moniulotteinen taulukko välitetään osoittimena nollanteen elementtiinsä. Esimerkissämme matrix:in tyyppi on osoitin kymmenen int-tyypin taulukkoon. Kuten taulukkoparametreilla, joilla on vain yksi ulottuvuus, moniulotteisen taulukon ensimmäinen ulottuvuus ei ole sopiva parametrityypiksi. Tyyppitarkistus parametrille, joka on moniulotteinen taulukko, muutoin varmistaa, että taulukkoparametrin kaikki ulottuvuudet (paitsi ensimmäisen) ovat samoja kuin taulukkoargumentissa.

Huomaa, että sulut ovat välttämättömiä *matrix-nimen ympärillä ja johtuu indeksioperaattorin korkeammasta sidontajärjestyksestä. Seuraava

```
int *matrix[ 10 ];
```

esittelee matrix:in kymmenen osoittimen taulukoksi int-tyyppeihin.

### 7.3.4 Abstraktien säiliötyyppien parametrit

Luvussa 6 esiteltyjä abstrakteja säiliötyyppejä voidaan myös käyttää funktioparametrien esittelyyn. Olisimme voineet määritellä esimerkiksi putValues()-funktion parametrityypillä vector<int> sisäisen taulukkotyypin sijasta.

Säiliötyyppi on varsinaisesti luokkatyyppi ja sillä on huomattavasti enemmän toimintoja kuin sisäisellä taulukkotyypillä. Esimerkiksi parametri, joka on tyyppiä vector<int>, tietää sisältämiensä elementtien lukumäärän. Näimme edellisessä alikohdassa, että jos funktiolla on taulukkoparametri, on taulukon ensimmäisen ulottuvuuden koko funktiolle tuntematon ja voi olla tarpeellista määritellä lisäparametri taulukon koon ilmaisemiseksi.

Kun käytämme parametrityyppiä vector<int>, voimme kiertää tämän rajoituksen. Voimme esimerkiksi muuttaa putValues()-funktion määrittelyä seuraavasti:

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

const lineLength = 12; // elementit riville
void putValues( vector<int> vec )
{
    cout << "( " << vec.size() << " )< ";
    for ( int i = 0; i < vec.size(); ++i ) {

        if ( i % lineLength == 0 && i )
            cout << "\n\t"; // rivi täynnä

        cout << vec[ i ];

        // erota kaikki muut paitsi viimeinen elementti
        if ( i % lineLength != lineLength-1 &&
         i != vec.size()-1 )
            cout << ", ";
    }
    cout << " >\n";
}
```

main()-funktio, joka käynnistää uuden putValues()-funktiomme, näyttää tältä:

```
void putValues( vector<int> );

int main() {
    int i, j[ 2 ];
    // sijoita muuttujiin i ja j jotkut arvot

    vector<int> vec1(1); // luo yhden elementin vektori
    vec1[0] = i;
    putValues( vec1 );

    vector<int> vec2;    // luo tyhjä vektori
    // lisää elementit vektoriin vec2
    for ( int ix = 0;
        ix < sizeof( j ) / sizeof( j[0] );
        ++ix )
        // vec2[ix] == j[ix]
        vec2.push_back( j[ix] );
    putValues( vec2 );

    return 0;
}
```

Huomaa, että putValues()-funktion parametri on arvovälitysparametri. Kun säiliötyyppinen parametri välitetään arvona, kopioidaan säiliö kaikkine elementteineen paikallisiksi kopioiksi kutsuttuun funktioon. Koska kopiointi voi olla melko tehotonta, on parempi esitellä säiliötyyppiset parametrit viittausparametreina. Kuinka muuttaisit putValues()-funktion määrittelyn parametrin?

Muista, että kun funktio ei muokkaa parametrinsa arvoa, on parempi esitellä parametri viittauksena const-tyyppiin. putValues()-funktion viittausparametri tulisi sen vuoksi esitellä seuraavasti:

```
void putValues( const vector<int> & ) { ...
```

### 7.3.5 Oletusargumentit

Oletusargumentti on argumentin arvo (vaikkakaan ei aina täysin sopiva), joka on määrätty olemaan valtaosassa tapauksista sopiva arvo parametrille. Oletusargumentti vapauttaa ohjelmoijan jokaisen pienen yksityiskohdan huomioimiselta funktion rajapinnan suunnittelussa.

Funktio voi määrittää oletusargumentin yhteen tai useampaan parametriinsa käyttämällä alustussyntaksia parametriluettelossa. Esimerkiksi funktio, joka luo ja alustaa kaksiulotteisen merkkitaulukon, jolla aiotaan simuloida näyttöpäätettä, voi määrittää oletusargumentit merkkinäytön korkeudelle, leveydelle ja taustan merkille:

```
char *screenInit( int height = 24, int width = 80,
        char background = ' ' );
```

Funktio, jossa on oletusargumentti parametrille, voidaan käynnistää tämän parametrin argumentilla tai ilman sitä. Jos argumentti annetaan, se korvaa oletusargumentin arvon; muussa tapauksessa käytetään oletusargumenttia. Jokainen seuraavista screenInit()-funktion kutsuista on oikein:

```
char *cursor;

// yhtä kuin screenInit(24,80,' ')
cursor = screenInit();

// yhtä kuin screenInit(66,80,' ')
cursor = screenInit(66);

// yhtä kuin screenInit(66,256,' ')
cursor = screenInit(66, 256);

cursor = screenInit(66, 256, '#');
```

Kutsun argumentit ratkaistaan niiden positioiden avulla ja oletusargumentteja käytetään *puuttuvien* loppupään argumenttien korvaamisessa funktion kutsussa. On esimerkiksi mahdotonta antaa merkkiarvoa argumenttina background-parametrille antamatta argumentteja height- ja width-parametreille.

```
// yhtä kuin screenInit('?',80,' ')
cursor = screenInit('?');

// virhe, ei yhtä kuin screenInit(24,80,'?')
cursor = screenInit( , , '?');
```

Osa funktion oletusargumenttien suunnittelusta on parametrien järjestelyä parametriluettelossa niin, että ne, jotka todennäköisemmin saavat käyttäjän määrittelemän arvon, esiintyvät ensin, ja ne, jotka todennäköisemmin käyttävät oletusarvoja, esiintyvät viimeisinä. screenInit()-funktion suunnittelussa oletetaan (ehkäpä kokemuksen kautta havaittu), että height on arvo, jonka käyttäjä todennäköisemmin antaa.

Funktion esittelyssä voidaan määrittää oletusargumentteja kaikille tai osalle sen parametreista. Oikeanpuoleisin alustamaton parametri pitää varustaa oletusargumentilla ennen kuin yhtäkään oletusargumenttia parametrista vasemmalle voidaan antaa. Tämä johtuu jälleen siitä, että funktion kutsun argumentit ratkaistaan niiden positioiden perusteella.

```
// virhe: width-parametrilla pitää olla oletusargumentti, ennen kuin
//        sellainen on määritetty height-parametrille
char *screenInit( int height = 24, int width,
                  char background = ' ' );
```

Parametrilla voi olla oletusargumentti määritettynä vain kerran tiedostossa. Esimerkiksi seuraava on virheellinen:

```
// ff.h
int ff( int = 0 );

// ff.C
#include "ff.h"
int ff( int i = 0 ) { ... } // virhe
```

Käytännön mukaisesti oletusargumentti on ilmaistu funktion esittelyssä, joka sisältyy julkiseen otsikkotiedostoon (siihen, joka kuvaa funktion rajapinnan) eikä funktion määrittelyyn. Jos oletusargumentti on annettu funktion määrittelyn parametriluettelossa, oletusargumentti on käytettävissä vain niissä funktion kutsuissa ja siinä tiedostossa, joka sisältää funktion määrittelyn.

Jälkeenpäin tehdyissä funktion esittelyissä voidaan määrittää lisää oletusargumentteja — kätevä menetelmä mukauttaa yleinen funktio tiettyyn sovelluskäyttöön. UNIX-järjestelmäkirjaston funktio chmod() muuttaa tiedoston suojaustasoa. Sen esittely löytyy järjestelmän otsikkotiedostosta <cstdlib>. Se on esitelty seuraavasti:

```
int chmod( char *filePath, int protMode );
```

protMode edustaa tiedostosuojauksen tilaa ja filePath tiedoston nimeä sekä sijaintipaikkaa polkuineen. Jos tietty sovellus aina muuttaa tiedoston suojauksen tilaan *vain-luku*, niin sen sijaan, että sovellus ilmaisisi sen joka kerta, chmod() voidaan esitellä uudelleen antamalla oletusarvo:

```
#include <cstdlib>
int chmod( char *filePath, int protMode=0444 );
```

Jos seuraava funktion esittely on annettu otsikkotiedostossa

```
int ff( int a, int b, int c = 0 ); // ff.h
```

kuinka voimme esitellä funktion ff() tiedostoomme niin, että voimme antaa b:n oletusargumenttina? Seuraava on virheellinen, koska se määrittää uudelleen c:n oletusargumentin:

```
#include "ff.h"
int ff( int a, int b = 0, int c = 0 ); // virhe
```

Seuraava näyttää myös virheelliseltä, mutta on itse asiassa virheetön uudelleenesittely:

```
#include "ff.h"
int ff( int a, int b = 0, int c ); // ok
```

Tässä vaiheessa ff()-funktion uudelleenesittelyä on b oikeanpuoleisin argumentti ilman oletusargumenttia. Siitä syystä sääntöä, jossa sijoitettava oletusargumentti olisi positioltaan oikeanpuoleisin, ei ole rikottu. Voimme itse asiassa esitellä ff():n nyt uudelleen kolmannen kerran:

```
#include "ff.h"

int ff( int a, int b = 0, int c ); // ok
int ff( int a = 0, int b, int c ); // ok
```

Oletusargumentin ei tarvitse olla vakiolauseke. Mitä tahansa lauseketta voidaan käyttää. Esimerkiksi:

```
int aDefault();
int bDefault( int );
int cDefault( double = 7.8 );

int glob;

int ff( int a = aDefault() ,
        int b = bDefault( glob ) ,
        int c = cDefault() );
```

Kun oletusargumentti on lauseke, se arvioidaan sillä hetkellä, kun funktiota kutsutaan. Esimerkiksi cDefault()-funktiota kutsutaan, jotta saadaan arvo c:lle joka kerta, kun ff()-funktiota kutsutaan ilman argumenttia.

### 7.3.6 Ellipsi eli kolme pistettä

Joskus on mahdotonta luetella kaikkien argumenttien tyyppejä ja lukumääriä, joita funktiolle voitaisiin välittää. Näissä tapauksissa voidaan määrittää *ellipsi* eli kolme pistettä (...) funktion parametriluettelossa.

Ellipsille ei tehdä tyyppitarkistusta. Sen läsnäolo kertoo kääntäjälle, että kun funktiota kutsutaan, voidaan olla antamatta tai annetaan argumentteja ja että niiden tyypit ovat tuntemattomia. Ellipsin yhteydessä käytetään kahta eri muotoa:

```
void foo( parm_list, ... );
void foo( ... );
```

Ensimmäisessä muodossa on esittelyitä tietylle määrälle funktion parametreja. Tässä tapauksessa tehdään tyyppitarkistus funktion kutsun yhteydessä argumenteille, jotka vastaavat eksplisiittisesti esiteltyjä parametreja, kun taas tyyppitarkistusta ei tehdä argumenteille, jotka vastaavat ellipsiä. Tässä ensimmäisessä muodossa parametriesittelyiden jälkeinen pilkku on valinnainen.

C-vakiokirjaston `printf()`-funktio on esimerkki tapauksesta, jolloin ellipsi on välttämätön. `printf()`-funktion ensimmäinen parametri on aina C-tyylinen merkkijono.

```
int printf( const char* ... );
```

Tämä edellyttää, että kun `printf()`-funktiota kutsutaan, on ensimmäisen argumentin aina oltava `const char*` -tyyppinen. Kun `printf()`-funktiota kutsutaan, päätellään ensimmäisen argumentin perusteella, jota myös muotoilumerkkijonoksi kutsutaan, onko merkkijonon jälkeen annettu muita argumentteja. Muotoilumerkkijonon metamerkit, jotka on eroteltu `%`-merkein, ilmaisevat lisäargumenttien mukanaolon. Esimerkiksi kutsussa

```
printf( "hello, world\n" );
```

on yksi merkkijonoargumentti. Kuitenkin kutsussa

```
printf( "hello, %s\n", userName );
```

on kaksi argumenttia. Merkki `%` ilmaisee toisen argumentin läsnäolon; `s` ilmaisee, että argumentin tyyppi on merkkijono.

Useimmat funktiot, jotka käyttävät ellipsiä, käyttävät hyväkseen parametrien eksplisiittisesti ilmoitettuja tietoja saadakseen funktion kutsussa annettujen argumenttien tyypit ja lukumäärän. Siitä syystä ensimmäinen muoto, jossa ellipsiä käytetään, on kaikkein yleisimmin käytetty.

Huomaa, että seuraavat kaksi esittelyä eivät ole samanarvoisia:

```
void f();
void f( ... );
```

Ensimmäisessä ilmentymässä f() on esitelty funktioksi, jolle ei anneta parametreja; toisessa ilmentymässä f() on esitelty funktioksi, jolle ei anneta yhtään tai annetaan yksi tai useampi argumentti. Kutsut

```
f( someValue );
f( cnt, a, b, c );
```

ovat sallittuja käynnistyksiä vain toiselle esittelylle. Kutsua

```
f();
```

voidaan käyttää joko ensimmäisen tai toisen funktion käynnistämiseen.

### Harjoitus 7.4

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

```
(a) void print( int arr[][], int size );
(b) int ff( int a, int b = 0, int c = 0 );
(c) void operate( int *matrix[] );
(d) char *screenInit( int height = 24, int width,
                      char background );
(e) void putValues( int (&ia)[] );
```

### Harjoitus 7.5

Jokainen näiden funktioiden uudelleenesittely on virheellinen. Miksi?

```
(a) char *screenInit( int height, int width,
                      char background = ' ' );
    char *screenInit( int height = 24, int width,
                      char background );

(b) void print( int (*arr)[6], int size );
    void print( int (*arr)[5], int size );

(c) void manip( int *pi, int first, int end = 0 );
    void manip( int *pi, int first = 0, int end = 0 );
```

### Harjoitus 7.6

Olkoon annettu seuraavat esittelyt. Mitkä niiden kutsuista ovat virheellisiä, vai onko yksikään? Miksi?

```
// esittelyt
void print( int arr[][5], int size );
void operate(int *matrix[7]);
char *screenInit( int height = 24, int width = 80,
                  char background = ' ' );

(a) screenInit(); // funktion kutsu

(b) int *matrix[5];
   operator( matrix ); // funktion kutsu

(c) int arr[5][5];
   print( arr, 5 ); // funktion kutsu
```

### Harjoitus 7.7

Kirjoita uudelleen putValues()-funktio, jossa käytettiin vektoria vector<int> ja esitettiin alikohdassa 7.3.4 niin, että se käsittelee sen sijaan listaa list<string>. Tulosta yksi merkkijono per rivi niin, että kahden merkkijonon lista tulostettaisiin näin:

```
( 2 )
<
"first string"
"second string"
>
```

Kirjoita main()-funktio, joka käynnistää tämän uuden putValues()-funktion merkkijonolistalle, joka sisältää seuraavat arvot:

```
"put function declarations in header files"
"use abstract container types instead of built-in arrays"
"declare class parameters as references"
"use reference to const types for invariant parameters"
"use less than eight parameters"
```

### Harjoitus 7.8

Milloin käyttäisit parametria, joka on osoitin? Milloin käyttäisit parametria, joka on viittaus? Perustele molempien edut ja haitat.

## 7.4 Arvon palauttaminen

return-lause sijoitetaan funktion runkoon. Tämä lause päättää funktion, joka on juuri suoritettavana. Kun return-lause kohdataan ohjelman suorituksen aikana, ohjelman kontrolli palaa funktioon, josta juuri nyt päättynyttä funktiota kutsuttiin. return-lauseella on kaksi muotoa:

```
return;
return lauseke;
```

Ensimmäistä muotoa käytetään funktiossa, jonka paluutyyppi on void. return-lause ei ole täysin välttämätöntä funktiossa, jonka paluutyyppi on void. Sitä käytetään pääasiassa funktion ennenaikaiseen päättämiseen. (Tällainen return-lauseen käyttö rinnastetaan break-lauseen käyttöön silmukan sisällä. break-lauseet on esitelty kohdassa 5.8.) Implisiittinen return tapahtuu funktion viimeisen lauseen jälkeen. Esimerkiksi:

```
void d_copy( double *src, double *dst, int sz )
{
    /* kopioi "src"-taulukko "dst"-taulukkoon
     * oletuksen yksinkertaistaminen: taulukot ovat samankokoisia
     */

    // jos jompikumpi osoittimista on 0, palaa
    if ( !src || !dst )
        return;

    // jos kaksi parametria viittaavat samaan taulukkoon, palaa
    if ( src == dst )
        return;

    // ei mitään kopioitavaa
    if ( sz == 0 )
        return;

    // yhä täällä? sitten on aika tehdä jotain työtä
    for ( int ix = 0; ix < sz; ++ix )
        dst[ix] = src[ix];

    // eksplisiittinen return ei ole välttämätöntä
    // palaa automaattisesti kutsuvaan funktioon
}
```

Toinen return-lauseen muoto sisältää funktion tuloksen. Tulos voi olla kuinka monimutkainen tahansa lauseke; se voi itse sisältää funktion kutsun. Toteutus factorial() esimerkiksi sisältää seuraavan return-lauseen (näemme factorial()-funktion toteutuksen seuraavassa kohdassa):

```
return val * factorial(val-1);
```

*Arvon palauttavan* funktion — jonka ei ole esitelty palauttavan void-tyyppiä — pitää palauttaa arvo. return-lauseen puuttuminen johtaa käännöksenaikaiseen virheeseen. Vaikka C++ ei

voi taata tuloksen oikeellisuutta, se takaa ainakin sen, että jokainen arvon palauttava funktio antaa tuloksen. Esimerkiksi seuraava ohjelma ei käänny, koska kaksi sen poistumispistettä eivät palauta arvoa.

```
// Matrix-luokan rajapinnan määrittely
#include "Matrix.h"

bool is_equal( const Matrix &m1, const Matrix &m2 )
{
    /* jos kahden Matrix-olion sisällöt ovat samat,
     * palauta arvo true;
     * muussa tapauksessa palauta arvo false
     */

    // vertaa sarakkeiden lukumäärää
    if ( m1.colSize() != m2.colSize() )
        // ohjelmavirhe: ei palauta arvoa
        return;

    // vertaa rivien lukumäärää
    if ( m1.rowSize() != m2.rowSize() )
        // ohjelmavirhe: ei palauta arvoa
        return;

    // käy kumpaakin Matrix-oliota läpi, kunnes jompikumpi on erisuuri
    // tai kaikki elementit on tutkittu
    for ( int row = 0; row < m1.rowSize(); ++row )
        for ( int col = 0; col < m1.colSize(); ++col )
            if ( m1[row][col] != m2[row][col] )
                return false;

    // ohjelmavirhe: ei palauta arvoa,
    // kun m1 == m2
}
```

Jos palautettu tyyppi ei ihan täsmällisesti vastaa funktion paluutyyppiä, tehdään implisiittinen konversio, jos se on mahdollista. Ellei implisiittinen konversio ole mahdollinen, käännöksenaikainen virheilmoitus generoituu. (Tyyppikonversioita käsiteltiin kohdassa 4.14.)

Oletuksena on, että funktion palauttama arvo *välitetään arvona*. Tämä tarkoittaa, että funktio, johon kontrolli palautuu, vastaanottaa kopion lausekkeen arvosta, joka on määritetty return-lauseeseen. Esimerkiksi:

```
Matrix grow( Matrix* p ) {
    Matrix val;
    // ...
    return val;
}
```

grow() palauttaa kopion val-arvosta kutsuvalle funktiolle. Kutsuva funktio ei voi millään tavalla muokata val-arvoa.

Tämä oletuskäyttäytyminen voidaan korvata. Funktio voidaan esitellä niin, että se palauttaa osoittimen tai viittauksen. Kun funktio palauttaa viittauksen, niin kutsuva funktio vastaanottaa *lvaluen* val-arvosta. Kutsuva funktio voi sitten muokata val-arvoa tai ottaa sen osoitteen. grow() voidaan esitellä palauttamaan viittaus seuraavasti:

```
Matrix& grow( Matrix* p ) {
    Matrix *res;
    // varaa suurempi Matrix dynaamisesta muistista
    // res viitaa tähän uuteen Matrix:iin
    // kopioi *p:n sisältö *res:iin
    return *res;
}
```

Jos paluuarvo on suuri luokkaolio, on paljon tehokkaampaa käyttää viittausta (tai osoitinta) paluutyyppinä kuin palauttaa luokkaolion arvo. Joissakin tapauksissa kääntäjä osaa automaattisesti muuntaa arvopalautuksen viittauspalautukseksi. Tätä optimointia kutsutaan nimellä *nimetyn paluuarvon optimointi* ja kuvataan kohdassa 14.8.

Ohjelmoijan tulisi olla tietoinen seuraavista kahdesta potentiaalisesta kompastuskivestä, kun funktio esitellään palauttamaan viittaus:

1. Palautetaan viittaus paikalliseen olioon. Paikallisen olion elinkaari päättyy funktion päättyessä. (Paikallisten olioiden elinkaarta käsitellään kohdassa 8.3.) Viittaus jää muistiin tuntemattomaan tilaan funktion päättymisen jälkeen. Esimerkiksi:

```
// ongelma: palauttaa viittauksen paikalliseen olioon
Matrix& add( Matrix &m1, Matrix &m2 )
{
    Matrix result;

    if ( m1.isZero() )
        return m2;
    if ( m2.isZero() )
        return m1;

    // laske yhteen kahden Matrix-olion sisällöt

    // hups: tulos viittaa kyseenalaiseen paikkaan paluun jälkeen
    return result;
}
```

   Tässä tapauksessa paluutyyppi tulisi esitellä muuna kuin viittauksena. Silloin paikallinen muuttuja ehditään kopioida ennen kuin paikallisen olion elinkaari päättyy:

```
Matrix add( ...
```

2. Funktio palauttaa lvaluen. Kaikki muutokset palautettuun arvoon muuttavat varsinaista palautettua oliota. Esimerkiksi:

```
#include <vector>

int &get_val( vector<int> &vi, int ix ) {
    return vi[ix];
}

int ai[4] = { 0, 1, 2, 3 };
vector<int> vec( ai, ai+4 ); // kopioi 4 elementtiä vektoriin vec

int main() {
    // kasvattaa vec[0]:n arvoon 1
    get_val( vec,0 )++;
    // ...
}
```

Jotta estettäisiin viittauksen paluuarvon tahaton muokkaus, tulisi paluuarvo esitellä const-tyyppiseksi:

```
const int &get_val( ...
```

Esimerkki siitä, kuinka lvalue palautetaan, kun tarkoituksena on muokata alkuperäistä oliota, esiteltiin kohdassa 2.3, kun käsiteltiin IntArray-luokan ylikuormitettua indeksioperaattoria.

### 7.4.1 Parametrit ja paluuarvot vastaan globaalit oliot

Ohjelman lukuisat funktiot voivat kommunikoida keskenään kahdella mekanismilla. (*Kommunikoinnilla* tarkoitamme arvojen vaihtoa.) Toinen metodi niistä käyttää globaaleja ohjelmaolioita ja toinen funktion parametriluetteloa ja paluuarvoa.

*Globaali olio* on määritelty funktion määrittelyn ulkopuolelle. Esimerkiksi:

```
int glob;
int main() {
    // mitä tahansa...
}
```

glob on globaali olio. (Luvussa 8 käsitellään lisää globaalia viittausaluetta ja globaaleja olioita.) Globaalin olion yleinen käytettävyys mistä tahansa ohjelman alueelta on samalla sen sekä pääetu että mitä merkittävin alttius virheille. Globaalin olion näkyvyys tekee siitä kätevän kommunikointimekanismin ohjelman eri osien välille. Globaalien olioiden käyttöön luottamisen haitat funktioiden välisessä kommunikoinnissa ovat seuraavat:

- Funktio, joka käyttää globaalia oliota, on riippuvainen tuon olion olemassaolosta ja tyypistä tehden tuon funktion uudelleenkäytön eri yhteyksissä vaikeammaksi.

- Jos ohjelmaa pitää muokata, globaaliset riippuvuudet kasvattavat virheiden todennäköisyyttä. Lisäksi paikallisten muutosten esittely vaatii koko ohjelman ymmärtämistä.
- Jos globaali olio saa virheellisen arvon, pitää koko ohjelma käydä läpi virheen löytämiseksi; lokalisointia ei ole.
- Rekursiota on vaikeampi saada onnistumaan, kun funktio käyttää globaalia oliota. (Rekursio tapahtuu, kun funktio kutsuu itseään. Katsomme rekursiota kohdassa 7.5.)
- Kun käytetään säikeitä (*threads*), vaaditaan erityistä koodausta globaalien olioiden lukemisen ja kirjoittamisen synkronointiin lukuisten säikeiden joukossa. Synkronoinnin puute on yleinen ohjelmointivirheen lähde säikeitä käytettäessä. (Katso artikkeli “Distributing Object Computing in C++”, Steve Vinoski ja Doug Schmidt julkaisusta [LIPPMAN96b], jossa on esimerkki säikeiden ohjelmoinnista C++:ssa.)

Tästä syystä suositellaan, että ohjelman funktiot kommunikoivat tietoa käyttäen parametriluetteloitaan ja paluuarvojaan.

Virheen todennäköisyys kasvaa argumenttien välityksessä funktiolle, kun parametriluettelon koko kasvaa. Yleissääntönä on, että parametreja tulisi olla enintään kahdeksan. Vaihtoehtona pitkälle parametriluettelolle ohjelmoija voi esitellä parametrin luokaksi, taulukoksi tai joksikin säiliötyypiksi; sellainen parametri voi sisältää joukon parametriarvoja.

Samalla tavalla funktio voi palauttaa vain yhden arvon. Jos ohjelman logiikka vaatii, että sen pitää palauttaa useita arvoja, ohjelmoija voi esitellä jotkut funktion parametrit viittauksiksi, jolloin funktio voi suoraan muokata vastaavia argumentteja ja siten asettaa argumentteihin joitakin “lisäpaluuarvoja” tai ohjelmoija voi esitellä funktion paluutyypillä, joka on luokka tai jokin säiliötyypeistä, joka sisältää joukon paluuarvoja.

### Harjoitus 7.9

Mitkä ovat return-lauseen kaksi eri muotoa? Selitä, milloin käyttäisit kumpaakin muotoa.

### Harjoitus 7.10

Minkä potentiaalisen suorituksenaikaisen ongelman näet seuraavassa funktion määrittelyssä?

```
vector<string> &readText( ) {
    vector<string> text;

    string word;
    while ( cin >> word ) {
        text.push_back( word );
        // ...
    }
```

```
    // ....
    return text;
}
```

**Harjoitus 7.11**

Kuinka palauttaisit useamman kuin yhden arvon funktiosta? Kuvaile ratkaisusi edut ja haitat.

## 7.5 Rekursio

Funktiota, joka kutsuu itseään joko suoraan tai epäsuoraan, kutsutaan *rekursiiviseksi funktioksi*. Seuraava rgcd()-funktio on rekursiivinen funktio:

```
int rgcd( int v1, int v2 )
{
    if ( v2 != 0 )
        return rgcd( v2, v1%v2 );
    return v1;
}
```

Rekursiivisen funktion pitää aina määritellä *pysäytysehto*; muussa tapauksessa funktio rekursoi "ikuisesti". Tätä kutsutaan joskus *päättymättömän rekursion* virheeksi. rgcd()-funktion tapauksessa ehto on jakojäännös 0.

Kutsu

```
rgcd( 15, 123 );
```

saa arvon 3. Taulukossa 7.1 seurataan suoritusta.

**Taulukko 7.1 Seurataan kutsua rgcd(15,123)**

| arvo1 | arvo2 | Paluu |
|---|---|---|
| 15 | 123 | rgcd( 123, 15 ) |
| 123 | 15 | rgcd( 15, 3 ) |
| 15 | 3 | rgcd( 3, 0 ) |
| 3 | 0 | 3 |

Viimeinen kutsu

```
rgcd(3,0);
```

täyttää pysäytysehdon. Se palauttaa suurimman yhteisen nimittäjän, 3. Tästä arvosta tulee aikaisempien peräkkäisten kutsujen paluuarvo. Arvon sanotaan *suodattuvan* nousevasti, kunnes suoritus palaa funktioon, joka kutsui rgcd()-funktiota ensimmäisen kerran.

Rekursiivisen funktion suoritus on todennäköisesti hitaampi kuin sen rekursioton (eli *iteratiivinen*) kollega, koska funktion käynnistykseen liittyy kuormitusta. Rekursiivinen funktio on

kuitenkin todennäköisesti pienempi ja helpompi ymmärtää.

Numeron n kertoma saadaan aikaan, kun lasketaan numeroiden 1 — n tulo. Numeron 5 kertoma on esimerkiksi 120.

```
1 x 2 x 3 x 4 x 5 = 120
```

Numeron kertoman laskenta voidaan toteuttaa rekursiivisella funktiolla:

```
unsigned long
factorial( int val ) {
    if ( val > 1 )
        return val * factorial( val-1 );
    return 1;
}
```

Pysäyttävä ehto on tässä tapauksessa se, kun val sisältää arvon 1.

**Harjoitus 7.12**

Kirjoita funktio factorial() uudelleen iteratiivisena funktiona.

**Harjoitus 7.13**

Mitä tapahtuisi, jos factorial()-funktion pysäyttävä ehto olisi seuraava?

```
if ( val != 0 )
```

## 7.6 Välittömät eli inline-funktiot

Mietitäänpä seuraavaa min()-funktiota:

```
int min( int v1, int v2 )
{
    return( v1 < v2 ? v1 : v2 );
}
```

Näinkin pienen funktion määrittelyyn liittyy seuraavaa:

- On yleensä helpompi lukea min()-kutsu ja tulkita sen tarkoitus kuin lukea ehdollisen operaattorin ilmentymä ja yrittää ymmärtää, mitä koodi tekee; etenkin, kun v1 ja v2 ovat monimutkaisia lausekkeita.
- On helpompi muuttaa yhtä lokalisoitua toteutusta kuin 300 esiintymää sovelluksesta. Jos esimerkiksi on päätetty, että testin pitäisi olla

  ```
  ( v1 == v2 || v1 < v2 )
  ```

  olisi jokaisen koodatun esiintymän löytäminen vaivalloista ja virhealtista.
- Merkitykset ovat yhdenmukaisia. On taattu, että jokainen testi toteutetaan samalla tavalla.

- Funktiota voidaan käyttää uudelleen sen sijaan, että se kirjoitettaisiin uudelleen muihin sovelluksiin.

min()-funktion kutsussa on kuitenkin eräs vakava varjopuoli: funktion kutsuminen on hitaampaa kuin suora ehdollisen operaattorin arviointi. Kaksi argumenttia pitää kopioida, koneen rekisterit pitää tallentaa ja ohjelman pitää haarautua uuteen paikkaan. Kovakoodattu ehdollinen operaattori on yksinkertaisesti nopeampi.

Välittömät (*inline*) funktiot ovat ratkaisu tähän. Funktio, joka määritetään välittömäksi, laajennetaan "rivilleen" jokaisessa kohdassa ohjelmaa, jossa se käynnistetään. Esimerkiksi

```
int minVal2 = min( i, j );
```

laajennetaan käännöksen aikana näin:

```
int minVal2 = i < j ? i : j;
```

Suorituksenaikainen min()-funktion kutsujen kuorma on siten poistettu.

min() esitellään välittömäksi inline-avainsanalla ennen funktion paluutyyppiä funktion esittelyssä tai määrittelyssä:

```
inline int min( int v1, int v2 ) { /* ... */ }
```

Huomaa, että välittömäksi määritys on vain ehdotus kääntäjälle. Kääntäjä voi ottaa huomioon tai olla ottamatta huomioon tätä ehdotusta, koska välittömäksi esitelty funktio ei ole hyvä ehdotus laajennusta varten tässä kutsukohdassa. Esimerkiksi rekursiivista funktiota kuten rgcd() ei voida täydellisesti laajentaa tässä kutsukohdassa (vaikka sen ensimmäinen käynnistys voidaan). 1200 rivin funktiotakaan tuskin laajennetaan kutsukohdassaan. Välittömien funktioiden mekanismi on tarkoitettu pienten, suoraviivaisten ja usein kutsuttujen funktioiden optimointiin. Se kuuluu tärkeimpiin tiedon piilotuksen tukimuotoihin abstraktien tietotyyppien suunnittelussa, kuten IntArray-luokassa, joka esiteltiin kohdassa 2.3 ja joka käytti välitöntä size()-jäsenfunktiota.

Välittömän funktion määrittelyn pitää olla näkyvissä kääntäjälle, jotta se voisi laajentaa funktion kutsukohdassa. Päinvastoin kuin tavalliset funktiot, välitön funktio pitää määritellä jokaisessa tekstitiedostossa, jossa funktiota kutsutaan. Välittömän funktion määrittelyjen, jotka esiintyvät eri tiedostoissa muodostaen ohjelman, pitää tietysti olla samanlaisia. Ohjelmaan, joka muodostuu kahdesta tiedostosta, compute.C ja draw.C, ei ohjelmoijan pidä määritellä välitöntä funktiota min() niin, että se merkitsisi yhtä asiaa tiedostossa compute.C ja jotain muuta tiedostossa draw.C. Elleivät nuo kaksi määrittelyä ole samanlaisia, ohjelman käyttäytyminen on tuntematon: ei ole varmuutta siitä, mitä monista eri määrittelyistä kääntäjä tulee käyttämään muille kuin välittömille funktiokutsuille, eikä ohjelma toimi odottamallasi tavalla.

Suositeltava tapa varmistua siitä, että tätä ei tapahdu, on sijoittaa välittömän funktion määrittely otsikkotiedostoon ja ottaa tämä mukaan jokaiseen tekstitiedostoon, jossa välitöntä funktiota kutsutaan. Tämä menettely takaa, että on olemassa vain yksi määrittely välittömälle funktiolle eivätkä ohjelmoijat tarpeettomasti kirjoita uudelleen koodia, joka voi johtaa myöhemmin tahattomaan funktioiden täsmäämättömyyteen ohjelman elinkaaren aikana.

Koska min() on yleinen operaatio, sen toteutus on tehty C++-vakiokirjastoon. min()-operaa-

tio on eräs geneerisistä algoritmeista, jotka on kuvattu luvussa 12, ja sen käyttöä on kuvattu liitteessä. Kirjastossa min() määritellään funktiomalliksi, joka mahdollistaa min()-operaation käytön muillakin aritmeettisilla operandeilla kuin int. Funktiomallit käsitellään luvussa 10.

## 7.7 Linkitysdirektiivit: extern "C"

Jos ohjelmoija haluaa kutsua funktiota, joka on kirjoitettu muulla ohjelmointikielellä — erityisesti C-kielellä — kääntäjälle pitää kertoa, että tulee käyttää erilaisia vaatimuksia funktiota kutsuttaessa. Esimerkiksi funktion nimi tai tapa, jolla argumentit järjestetään funktion kutsussa, voivat olla erilaisia, jos kutsutaan C++-funktiota tai jollain muulla ohjelmointikielellä kirjoitettua funktiota.

Ohjelmoija ilmaisee kääntäjälle, että funktio on kirjoitettu eri ohjelmointikielellä, käyttämällä *linkitysdirektiiviä*. Linkitysdirektiivillä on kaksi eri muotoa. Se voi olla *yhden lauseen* muoto tai *yhdistetyn lauseen* muoto:

```
// yhden lauseen linkitysdirektiivi
extern "C" void exit(int);

// yhdistetyn lauseen linkitysdirektiivi
extern "C" {
    int printf( const char* ... );
    int scanf( const char* ... );
}

// yhdistetyn lauseen linkitysdirektiivi
extern "C" {
#include <cmath>
}
```

Linkitysdirektiivin ensimmäinen muoto muodostuu extern-avainsanasta ja merkkijonoliteraalista, jonka jälkeen tulee “normaali” funktion esittely. Vaikka funktio on kirjoitettu toisella kielellä, sen kutsut silti tyyppitarkistetaan täysin. Kääntäjä esimerkiksi tarkistaa, että exit()-funktiolle välitetyn argumentin tyyppi on int tai se voidaan implisiittisesti konvertoida int-tyyppiseksi.

Linkitysdirektiivin toisen muodon, yhdistetyn lauseen, aaltosulkujen sisälle voidaan laittaa useita funktion esittelyitä. Aaltosulut toimivat erottimina, jotka ilmaisevat, mitä esittelyitä linkitysdirektiivit koskevat. Muutoin aaltosulut jätetään huomiotta ja niiden sisällä esitellyt funktioiden nimet ovat näkyviä aivan kuin ne olisi esitelty yhdistetyn lauseen ulkopuolella. Esimerkiksi yhdistetty lause extern "C" edellisessä esimerkissä ilmaisee, että funktiot printf() ja scanf() on kirjoitettu C-ohjelmointikielellä. Muutoin esittelyiden merkitys on sama kuin jos printf() ja scanf() olisi esitelty yhdistetyn lauseen, extern "C", ulkopuolella.

Kun #include-direktiivi on linkitysdirektiivin yhdistetyn lauseen aaltosulkujen sisällä, kaikkien otsikkotiedostossa esiteltyjen funktioiden katsotaan kirjoitetun linkitysdirektiivin ohjelmointikielellä. Edellisessä esimerkissä funktiot, jotka on esitelty otsikkotiedostossa <cmath>,

ovat C-funktioita.

Linkitysdirektiivi ei voi olla funktion rungossa. Seuraava koodikatkelma on virheellinen ja aiheuttaa käännöksenaikaisen virheilmoituksen:

```
int main()
{
    // virhe: linkitysdirektiivi ei voi olla funktion sisällä
    extern "C" double sqrt( double );
    double getValue(); //ok

    double result = sqrt ( getValue() );
    //...
    return 0;
}
```

Jos siirrämme linkitysdirektiivin funktion rungon ulkopuolelle, ohjelma kääntyy virheittä:

```
extern "C" double sqrt( double );

int main()
{
    double getValue(); //ok

    double result = sqrt ( getValue() );
    //...
    return 0;
}
```

Linkitysdirektiivi olisi parempi sijoittaa otsikkotiedostoon, johon funktion rajapintaa kuvaava funktion esittely kuuluu.

Mitä, jos haluamme saada C++-funktion C-ohjelman käyttöön? Kuinka teemme sen? Käytämme jälleen `extern "C"` -linkitysdirektiiviä saadaksemme C++-funktion C-ohjelman käytettäväksi. Esimerkiksi:

```
// calc()-funktiota voidaan kutsua C-ohjelmista
extern "C" double calc( double dparm ) { /* ... */ }
```

Jos funktio on esitelty useammin kuin kerran samassa tiedostossa, voi linkitysdirektiivi esiintyä jokaisessa esittelyssä. Se voi myös esiintyä vain funktion ensimmäisessä esittelyssä, jolloin toinen ja seuraavat esittelyt saavat ensimmäisen esittelyn linkitysdirektiivin. Esimerkiksi:

```
// ---- myMath.h ----
extern "C" double calc( double );

// ---- myMath.C ----
// calc() esitelty otsikkotiedostossa myMath.h
#include "myMath.h"

// määrittelee extern "C" calc()-funktion
```

```
// calc()-funktiota voidaan kutsua C-ohjelmista
double calc( double dparm ) { // ...
```

Tässä kohdassa näimme esimerkkejä vain linkitysmäärityksille C-kielelle: extern "C". extern "C" on ainoa linkitysmääritys, jota kaikki C++-toteutukset tukevat. Toteutuksessa voi olla muita linkitysmäärityksiä kielille, joita ympäristössä käytetään yleisesti. Esimerkiksi linkitysmääritystä extern "Ada" voidaan käyttää esiteltäessä funktioita, jotka on kirjoitettu Ada-kielellä, extern "FORTRAN" funktioille, jotka on kirjoitettu FORTRAN-kielellä jne. Koska muut linkitysmääritykset ovat toteutuskohtaisia, suosittelemme, että tutkit toteutuksesi käyttäjäopasta saadaksesi lisätietoja muista linkitysmäärityksistä, joita se voi tarjota.

Tässä kohdassa esiteltiin extern-avainsanan ensimmäinen käyttötapa C++:ssa. Kohdassa 8.2 näemme lisää extern-avainsanan käyttötapoja olioiden ja funktioiden esittelyissä.

### Harjoitus 7.14

exit(), printf(), malloc(), strcpy() ja strlen() ovat C-kielen kirjastorutiineja. Muokkaa seuraavaa C-ohjelmaa niin, että se kääntyy ja linkittyy C++-ympäristössä.

```
const char *str = "hello";

void *malloc( int );
char *strcpy( char *, const char * );
int printf( const char *, ... );
int exit( int );
int strlen( const char * );

int main()
{   /* C-kielinen ohjelma */

    char* s = malloc( strlen(str)+1 );
    strcpy( s, str );
    printf( "%s, world\n", s );
    exit( 0 );
}
```

## 7.8 main(): komentorivin valitsimien käsittely ◆

Usein, kun suoritamme ohjelmiamme, välitämme niille komentorivin valitsimia. Voisimme esimerkiksi kirjoittaa

```
prog -d -o ofile data0
```

Itse asiassa komentorivin valitsimet ovat argumentteja main()-funktiolle ja niitä voidaan käsitellä main()-funktiossa C-tyylisen argv-nimisen merkkijonotaulukon kautta. Tässä kohdassa kuvaamme, kuinka komentorivin valitsimia tuetaan.

Ennen tätä kohtaa kaikki main()-funktiomme määrittelyt on esitelty tyhjällä parametriluettelolla:

```
int main() { ... }
```

main()-funktion laajennettu merkintätapa mahdollistaa, että voimme käsitellä käyttäjän määrittämiä valitsimia komentoriviltä, jos niitä on:

```
int main( int argc, char *argv[] ) { ... }
```

argc sisältää komentorivin valitsimien lukumäärän. argv sisältää argc kappaletta C-tyylisiä merkkijonoja, jotka erotellaan tyhjin merkein komentorivin valitsimiksi. Olkoon esimerkiksi annettu komentoriville

```
prog -d -o ofile data0
```

argc asetetaan arvoon 5 ja argv saa arvokseen seuraavat C-tyyliset merkkijonot:

```
argv[ 0 ] = "prog";
argv[ 1 ] = "-d";
argv[ 2 ] = "-o";
argv[ 3 ] = "ofile";
argv[ 4 ] = "data0";
```

argv[0] saa aina arvokseen käynnistetyn komennon. Indeksit 1 — argc-1 edustavat varsinaisia komennolle välitettyjä valitsimia.

Katsotaanpa, kuinka erottelemme ja arvioimme komentorivin valitsimet, jotka on tallennettu argv:hen. Esimerkissämme tuemme seuraavaa käyttötapaa:

```
program_name [-d] [-h] [-v]
        [-o output_file] [-l limit_value]
        file_name
        [ file_name [file_name [ ... ]]]
```

Kaikki hakasulkujen sisällä on valinnaista. Siten esimerkiksi minimaalisin komentorivi ilmaisee yksinkertaisesti vain käsiteltävän tiedoston:

```
prog chap1.doc
```

Vaihtoehtoisiin käynnistyksiin kuuluvat seuraavat:

```
prog -l 1024 -o chap1-2.out chap1.doc chap2.doc
prog -d chap3.doc
prog -l 512 -d chap4.doc
```

Komentorivin valitsimien käsittelyn perusvaiheet ovat seuraavat:

1. Erota jokainen valitsin vuorollaan argv:stä. Käytämme tähän for-silmukkaa ja alamme silmukoinnin indeksistä 1 (ja siten hyppäämme ohjelman nimen yli):

```
for ( int ix = 1; ix < argc; ++ix ) {
    char *pchar = argv[ ix ];
    // ...
}
```

2. Päättele valitsimen tyyppi. Jos se alkaa yhdysviivalla, silloin tiedämme, että se on joku näistä: { h, d, v, l, o }. Muussa tapauksessa se on joko varsinainen raja-arvo (limit), johon liittyy -l, tulostustiedoston nimi, johon liittyy -o tai tiedoston nimi, joka ohjelman pitää käsitellä. Käytämme switch-lausetta päätelläksemme, onko yhdysviiva läsnä:

```
    switch ( pchar[ 0 ] ) {
        case '-': {
            // havaitse -h, -d, -v, -l, -o
        }

        default: {
            // käsittele raja-arvo, joka tulee tämän jälkeen: -l
            //       tulostustiedosto, joka tulee tämän jälkeen: -o
            //       tiedostojen nimet ...
        }
    }
```

3. Kirjoita kohdan 2 käsittelyt.

Jos yhdysmerkki on läsnä, silloin siirrymme yksinkertaisesti seuraavaan merkkiin, jotta voimme päätellä käyttäjän määrittämän valitsimen. Seuraavassa on toteutuksemme tuon osan yleinen hahmotelma:

```
case '-': {
    switch( pchar[ 1 ] )
    {
        case 'd':
            // käsittele virheenetsintä, debug
            break;

        case 'v':
            // käsittele versiopyyntö
            break;

        case 'h':
            // käsittele helppi
            break;

        case 'o':
            // valmistaudu käsittelemään tulostustiedosto, output_file
            break;

        case 'l':
            // valmistaudu käsittelemään raja-arvo, limit_value
            break;

        default:
            // tunnistamaton valitsin:
            // raportoi siitä ja poistu
    }
}
```

Valitsin `-d` ottaa käyttöön virheenetsinnän eli *debuggauksen*. Jotta voimme käsitellä sen, asetamme olion

```
bool debug_on = false;
```

arvoon true:

```
case 'd':
    debug_on = true;
    break;
```

Ohjelmamme voi sisältää koodia kuten seuraavassa:

```
if ( debug_on )
    display_state_elements( obj );
```

Valitsin -v näyttää ohjelmamme versionumeron ja poistuu:

```
case 'v':
    cout << program_name << "::"
        << program_version << endl;
    return 0;
```

Valitsin -h generoi ohjelman usage()-ilmoituksen ja poistuu sitten (poistuminen tapahtuu usage()-funktiossa):

```
case 'h':
    // break ei ole välttämätön: poistuminen tapahtuu usage()-funktiossa
    usage();
```

Valitsin -o ilmaisee, että sen jälkeen tulee käyttäjän määrittämä tulostustiedoston nimi. Samalla tavalla valitsin -l ilmaisee, että raja-arvo tulee sen jälkeen. Kuinka meidän tulisi käsitellä se?

Ellei yhdysviivaa ole annettu, tiedämme, että meillä on joko raja-arvo, käyttäjän määrittämä tulostustiedosto tai käsiteltävän tiedoston nimi. Jotta voimme erottaa nämä kolme mahdollisuutta, asetamme olioita, jotka muistavat sisäiset tilat arvona true

```
// jos arvo on true, seuraava argumentti on tulostustiedosto
bool ofile_on = false;

// jos arvo on true, seuraava argumentti on raja-arvo
bool limit_on = false;
```

ja sijoitamme ne toteutuksemme valitsinten käsittelyosaan:

```
case 'l':
    limit_on = true;
    break;

case 'o':
    ofile_on = true;
    break;
```

Kun kohtaamme argumentin, joka ei ala yhdysviivalla, testaamme olioiden tilan, jotta voimme päätellä, mitä valitsin edustaa:

```
// joko a limit_value, output_file tai file_name
default: {
```

```
    // ofile_on asetetaan, jos havaittu -o
    if ( ofile_on ) {
        // käsittele output_file
        // ota ofile_on pois käytöstä
    }
    else
    if ( limit_on ) { // jos -l havaittu
        // käsittele limit_value
        // ota limit_on pois päältä
    }
    else {
        // käsittele file_name
    }
}
```

Jos argumentti on tulostustiedosto, asetamme ofile_on-tilan uudelleen arvoon false ja otamme tiedoston nimen:

```
if ( ofile_on ) {
    ofile_on = false;
    ofile = pchar;
}
```

Jos argumentti on raja-arvo, pitää C-tyylinen merkkijono muuntaa numeeriseen esitystapaan. Teemme sen vakiokirjaston funktiolla atoi(); se saa argumenttinaan C-tyylisen merkkijonon ja palauttaa int-arvon (on olemassa myös atof(), joka palauttaa float-arvon). Jotta voisimme käyttää atoi()-funktiota, otamme mukaan ctype.h-otsikkotiedoston. Pitää myös varmistua, että raja-arvo on positiivinen, ja asettaa limit_on arvoon false:

```
// int limit;

else
if ( limit_on ) {
    limit_on = false;
    limit = atoi( pchar );
    if ( limit < 0 ) {
        cerr << program_name << "::"
            << program_version << " : error: "
            << "negative value for limit.\n\n";
        usage( -2 );
    }
}
```

Muussa tapauksessa, ellei kummankaan tilaolion arvo ole tosi, meillä on tiedosto avattavana käsittelyä varten. Tallennamme sen nimen string-vektoriin:

```
else
    file_names.push_back( string( pchar ));
```

Kun käsittelemme komentoriviä, on suunnittelun ehkä merkittävin näkökohta valitsemamme tapa, jolla käsittelemme kelvottomat valitsimet. Teimme esimerkiksi negatiivisen rajaarvon antamisesta vakavan virheen. Se voi olla sopivaa tai sitten ei. Vaihtoehtoisesti olisimme havainneet sen olevan rajojensa ulkopuolella, varoittaneet käyttäjää ja asettaneet arvon uudelleen nollaksi tai johonkin merkityksellisempään oletusarvoon.

Toteutuksemme kaksi heikkoutta tulee ilmeiseksi, kun käyttäjämme kirjoittavat satunnaisesti tyhjiä merkkejä komentorivin valitsimien väliin. Esimerkiksi seuraavista ei kumpaakaan käsitellä:

```
prog - d data01
prog -oout_file data01
```

(Jätämme molemmat harjoitukseksi tämän kohdan loppuun.)

Tässä on ohjelmamme koko toteutus. (Olemme lisänneet tulostuskomentoja kuvataksemme sen käsittelyn etenemistä)

```
#include <iostream>

#include <string>
#include <vector>

#include <ctype.h>

const char *const program_name = "comline";
const char *const program_version = "version 0.01 (08/07/97)";

inline void usage( int exit_value = 0 )
{
    // tulostaa muotoillun käyttötapailmoituksen
    // ja poistuu käyttäen arvoa exit_value ...

    cerr << "usage:\n"
        << program_name << " "
        << "[-d] [-h] [-v] \n\t"
        << "[-o output_file] [-l limit] \n\t"
        << "file_name\n\t[file_name [file_name [ ... ]]]\n\n"
        << "where [] indicates optional option:\n\n\t"
        << "-h: help.\n\t\t"
        << "generates this message and exits\n\n\t"
        << "-v: version.\n\t\t"
        << "prints version information and exits\n\n\t"
        << "-d: debug.\n\t\tturns debugging on\n\n\t"
        << "-l limit\n\t\t"
        << "limit must be a non-negative integer\n\n\t"
        << "-o ofile\n\t\t"
        << "file within which to write out results\n\t\t"
        << "by default, results written to standard output\n\n"
        << "file_name\n\t\t"
        << "the name of the actual file to process\n\t\t"
```

```
            << "at least one file_name is required --\n\t\t"
            << "any number may be specified\n\n"
            << "examples:\n\t\t"
            << "$command chapter7.doc\n\t\t"
            << "$command -d -l 1024 -o test_7_8 "
            << "chapter7.doc chapter8.doc\n\n";

        exit( exit_value );
}

int main( int argc, char* argv[] )
{
        bool debug_on = false;
        bool ofile_on = false;
        bool limit_on = false;
        int limit = -1;

        string ofile;
        vector<string, allocator> file_names;

        cout << "illustration of handling command line arguments:\n"
             << "argc: " << argc << endl;

        for ( int ix = 1; ix < argc; ++ix )
        {
            cout << "argv[ " << ix << " ]: "
                 << argv[ ix ] << endl;

            char *pchar = argv[ ix ];
            switch ( pchar[ 0 ] )
            {
                case '-':
                {
                    cout << "case \'-\' found\n";
                    switch( pchar[ 1 ] )
                    {
                        case 'd':
                            cout << "-d found: "
                                 << "debugging turned on\n";

                            debug_on = true;
                            break;

                        case 'v':
                            cout << "-v found: "
                                 << "version info displayed\n";

                            cout << program_name
                                 << " ::  "
```

```
                    << program_version
                    << endl;

                return 0;

            case 'h':
                cout << "-h found: "
                    << "help information\n";

                // break ei välttämätön: usage() exits
                usage();

            case 'o':
                cout << "-o found: output file\n";
                ofile_on = true;
                break;

            case 'l':
                cout << "-l found: "
                    << "resource limit\n";

                limit_on = true;
                break;

            default:
                cerr << program_name
                    << " : error : "
                    << "unrecognized option: - "
                    << pchar << "\n\n";

                // break ei ole välttämätön: poistuminen tapahtuu usage()-funktiossa
                usage( -1 );
        }
        break;
    }

    default: // joko tiedoston nimi
        cout << "default nonhyphen argument: "
            << pchar << endl;

        if ( ofile_on ) {
            ofile_on = false;
            ofile = pchar;
        }
        else
        if ( limit_on ) {
            limit_on = false;
            limit = atoi( pchar );
            if ( limit < 0 ) {
```

```
                        cerr << program_name
                            << " : error : "
                            << "negative value for limit.\n\n";

                        usage( -2 );
                    }
                }
                else file_names.push_back( string( pchar ));
                break;
            }
        }

        if ( file_names.empty() ) {
            cerr << program_name
                << " : error : "
                << "no file specified for processing.\n\n";

            usage( -3 );
        }

        if ( limit != -1 )
            cout << "User-specifed limit: "
                << limit << endl;

        if ( ! ofile.empty() )
            cout << "User-specified output file: "
                << ofile << endl;

        cout << (file_names.size() == 1 ? "File " : "Files ")
            << "to be processed are the following:\n";

        for ( int inx = 0; inx < file_names.size(); ++inx )
            cout << "\t" << file_names[ inx ] << endl;
}
```

Tässä kokeillaan ohjelmaamme:

```
a.out -d -l 1024 -o test_7_8 chapter7.doc chapter8.doc
```

Tässä on komentorivin valitsimien käsittelyn seurantaa:

```
illustration of handling command line arguments:
argc: 8
argv[ 1 ]: -d
case '-' found
-d found: debugging turned on
argv[ 2 ]: -l
case '-' found
-l found: resource limit
argv[ 3 ]: 1024
default nonhyphen argument: 1024
```

```
argv[ 4 ]: -o
case '-' found
-o found: output file
argv[ 5 ]: test_7_8
default nonhyphen argument: test_7_8
argv[ 6 ]: chapter7.doc
default nonhyphen argument: chapter7.doc
argv[ 7 ]: chapter8.doc
default nonhyphen argument: chapter8.doc
User-specifed limit: 1024
User-specified output file: test_7_8
Files to be processed are the following:
      chapter7.doc
      chapter8.doc
```

### 7.8.1 Komentoriviluokka

Komentorivin valitsimien käsittelyn yksityiskohdat on parasta kapseloida niin, että emme sekoita main()-funktiota. Eräs kapseloinnin strategia on tietysti tehdä funktio. Esimerkiksi:

```
extern int parse_options( int arg_count,
                          char **arg_vector );

int main( int argc, char *argv[] ) {
    // ...
    int option_status;
    option_status = parse_options( argc, argv );
    // ...
}
```

Suunnittelua koskeva kysymys kuuluu, kuinka palauttaa käyttäjän välittämät arvot. Tyypillisesti nämä arvot on määritelty globaaleiksi olioiksi eikä niitä välitetä funktioon tai takaisin funktiosta. Vaihtoehtoisesti voimme kapseloida käsittelyn luokkaan.

Luokan tietojäsenet ovat olioita, jotka edustavat mahdollisia arvoja, joita käyttäjät asettavat. Joukolla julkisia, välittömiä jäsenfunktioita päästään käsittelemään näitä arvoja. Muodosta- ja alustaa ne oletusarvoikseen. Funktio saa argumentit argc ja argv sekä käsittelee valitsimet:

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

class CommandOpt {
public:
    CommandOpt() : _limit( -1 ), _debug_on( false ) {}
    int parse_options( int argc, char *argv[] );

    string out_file() { return _out_file; }
    bool   debug_on() { return _debug_on; }
    int    files()    { return _file_names.size(); }
```

```
        // tiedostonimien käsittelyyn (_file_names)
        string& operator[]( int ix );

    private:
        inline int usage( int exit_value = 0 );

        bool _debug_on;
        int _limit;
        string _out_file;
        vector<string, allocator> _file_names;

        static const char *const program_name;
        static const char *const program_version;
    };
```

Tässä on uudistettu main()-funktiomme[1]:

```
#include "CommandOpt.h"

int main( int argc, char *argv[] ) {
    // ...
    CommandOpt com_opt;
    int option_status;
    option_status = com_opt.parse_options(argc,argv);
    // ...
}
```

### Harjoitus 7.15

Lisää valitsimien -t (joka ottaa käyttöön ajastimen) ja -b (joka saa argumentin bufsize) käsittely. Varmistu, että päivität myös usage()-funktion. Esimerkiksi:

```
prog -t -b 512 data0
```

### Harjoitus 7.16

Nykyinen toteutuksemme ei pysty käsittelemään tilannetta, jossa valitsimen ja siihen liittyvän arvon välissä ei ole tyhjää merkkiä. Ihannetapauksessa hyväksyisimme valitsimen tyhjän merkin kanssa tai ilman. Muokkaa toteutustamme, että se toimii niin.

### Harjoitus 7.17

Nykyinen toteutuksemme ei pysty käsittelemään käyttäjän virhettä, jossa yhdysviivan ja valitsimen välissä on tyhjä merkki kuten tässä:

```
prog - d data0
```

1. CommandOpt-luokan täydellinen toteutus on saatavilla Addison–Wesleyn Web-paikasta.

Muokkaa toteutustamme niin, että se huomaa sen ja antaa siitä virheilmoituksen.

**Harjoitus 7.18**

Nykyinen toteutuksemme ei huomaa, jos valitsimia -l tai -o on useampia ilmentymiä. Muokkaa toteutustamme niin, että se huomaa sen. Millainen menettelypolitiikan tulisi olla?

**Harjoitus 7.19**

Toteutuksemme generoi vakavan virheen, jos käyttäjä määrittää tuntemattoman valitsimen. Onko se mielestäsi kohtuullista? Mitä muuta voisimme tehdä?

**Harjoitus 7.20**

Lisää tuki valitsimille, jotka alkavat plusmerkillä (+), joka tarkoittaa valitsimien +s ja +pt kuten myös +sp ja +ps käsittelyä. Oletetaan, että +s ottaa tarkan käsittelyn käyttöön ja että +p tukee aikaisempia rakenteita, jotka ovat nyt vanhentuneet. Esimerkiksi:

```
prog +s +p -d -b 1024 data0
```

## 7.9 Osoittimet funktioihin ◆

Oletetaan, että meitä on pyydetty tekemään lajittelufunktio, jota kutsutaan seuraavasti:

```
sort( start, end, compare );
```

start ja end ovat osoittimia merkkijonotaulukon elementteihin. sort()-funktio lajittelee taulukon elementit väliltä start ja end. compare määrittelee vertailuoperaation, jota käytetään kahden merkkijonon vertailuun taulukossa.

Mitä toteutusta tulisi käyttää compare-operaatioon? Voisimme lajitella taulukon merkkijonot aakkosjärjestykseen — tarkoittaa tapaa, jolla sanat on lajiteltu hakemistossa; tai voisimme lajitella ne pituuden perusteella niin, että lyhyemmät merkkijonot sijoitetaan taulukon alkuun ja pitemmät loppuun. Tarvitaan jokin piirre vaihtoehtoisten vertailuoperaatioiden määritykseen.

(Huomaa, että luvussa 12 kuvataan sort()-funktio ja muut C++-vakiokirjaston yleiset algoritmit. Tässä kohdassa, kun haluamme kuvata funktio-osoittimien käyttöä, kirjoitamme oman sort()-funktion, joka on yksinkertaistettu versio C++-vakiokirjaston funktiosta.)

Eräs strategia ratkaista nämä vaatimukset on tehdä kolmannesta argumentista, compare, funktio-osoitin, joka määrittää käytettävän vertailufunktion.

Jotta voisimme yksinkertaistaa sort()-funktion käyttöä menettämättä sen joustavuutta, voimme määrittää oletusarvoisen vertailuoperaation, jota käytettäisiin valtaosassa tapauksista. Olettakaamme, että yleisin tapaus on lajitella merkkijonot aakkosjärjestykseen ja että oletusargumentti määrittäisi vertailuoperaation, joka käyttäisi compare()-funktiota merkkijonoille (tämä funktio on ensimmäisen kerran esitelty kohdassa 6.10).

Tällä kertaa mietimme, kuinka toteuttaisimme sort()-funktiomme funktio-osoittimia käyttäen.

### 7.9.1 Osoittimen funktioon tyyppi

Kuinka osoitin funktioon voidaan esitellä? Miltä näyttää parametri, jolle annetaan argumenttina osoitin funktioon? Tässä on lexicoCompare()-funktion määrittely, joka vertailee kahta merkkijonoa aakkosjärjestyksessä:

```
#include <string>
int lexicoCompare( const string &s1, const string &s2 ) {
    return s1.compare(s2);
}
```

Jos kaikki merkit merkkijonoissa s1 ja s2 ovat yhtäsuuria, silloin lexicoCompare() palauttaa arvon 0; jos ensimmäisen parametrin merkkijono on pienempi kuin toisen parametrin, palautetaan negatiivinen numero; jos se on suurempi, palautetaan positiivinen numero.

Funktion nimi ei ole osa sen tyyppiä. Funktion tyyppi päätellään vain sen paluutyypin ja parametriluettelon perusteella. Osoittimen lexicoCompare()-funktioon pitää osoittaa samantyyppiseen funktioon (jossa on sama paluutyyppi ja samanlainen parametriluettelo) kuin lexicoCompare(). Yrittäkäämme:

```
int *pf( const string &, const string & ); // hups: ei ihan näin
```

Tämä on melkein oikein. Ongelma on, että kääntäjä tulkitsee lauseen esittelyksi funktiosta nimeltään pf, joka saa kaksi argumenttia ja palauttaa osoitintyypin int*. Parametriluettelo on oikein, mutta paluutyyppi ei ole ihan sitä, mitä haluamme. Käänteisoperaattori (*) liittyy paluutyyppiin tässä tapauksessa tyyppimääreellä int eikä pf. Sulut ovat välttämättömiä, jotta käänteisoperaattoriin liittyy nimenomaan pf:

```
int (*pf)( const string &, const string & ); //ok: tämä toimii
```

Tämä lause esittelee pf:n osoittimeksi funktioon, joka saa kaksi parametria ja palauttaa int-tyypin; tämä tarkoittaa osoitinta funktioon, jonka tyyppi on sama kuin lexicoCompare()-funktion.

Seuraavalla funktiolla on samanlainen tyyppi kuin lexicoCompare()-funktiolla, ja kumpaakin niistä voidaan osoittaa pf-osoittimella:

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

calc() ja gcd() ovat kuitenkin funktioita, joiden tyypit ovat erilaisia kuin kahden edellisen funktion, eikä pf voi osoittaa niitä:

```
int calc( int , int );
int gcd( int , int );
```

Osoitin, joka voisi osoittaa kumpaakin näistä funktioista, voidaan määritellä seuraavasti:

```
int (*pfi)( int, int );
```

Ellipsi on osa funktion tyyppiä. Kahdella funktiolla, joilla on muuten samanlaiset parametriluettelot, paitsi että toisella funktiolla on ellipsi eli kolme pistettä parametriluettelonsa lopussa, on erilaiset funktiotyypit. Osoittimilla sellaisiin funktioihin on erilaiset tyypit.

```
int printf( const char*, ... );
int strlen( const char* );
```

```
int (*pfce)( const char*, ... ); // ei voi osoittaa funktioon printf()
int (*pfc)( const char* );       // ei voi osoittaa funktioon strlen()
```

On olemassa yhtä monta selkeästi erotettavaa funktiotyyppiä kuin on olemassa selkeästi erotettavia funktioiden paluutyyppejä ja parametriluetteloita.

### 7.9.2 Alustus ja sijoitus

Muista, että taulukon nimeä pidetään osoittimena sen ensimmäiseen elementtiin, kun taulukon nimeen ei liity indeksioperaattoria. Funktion nimeä, kun siihen ei liity kutsuoperaattoria, pidetään osoittimena sen tyyppiseen funktioon. Esimerkiksi lauseke

```
lexicoCompare;
```

arvioidaan osoitintyypiksi

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

Kun osoiteoperaattoria käytetään funktion nimeen, se johtaa myös tuon funktion tyyppiseksi osoittimeksi funktioon. Siten sekä `lexicoCompare` että `&lexicoCompare` omaavat saman tyypin. Osoitin funktioon voidaan alustaa seuraavasti:

```
int (*pfi)( const string &, const string & ) = lexicoCompare;
int (*pfi2)( const string &, const string & ) = &lexicoCompare;
```

Funktio-osoittimeen voidaan sijoittaa arvo, kuten seuraavassa:

```
pfi = lexicoCompare;
pfi2 = pfi;
```

Alustus tai sijoitus on sallittu vain, jos sijoitusoperaattorin vasemman puolen osoittimen parametriluettelo ja paluutyyppi ovat täsmälleen samanlaisia kuin sijoitusoperaattorin oikealla puolella olevan funktion tai osoittimen parametriluettelo ja paluutyyppi. Elleivät ne täsmää, aiheutuu käännöksenaikainen virheilmoitus; implisiittistä tyyppikonversiota ei ole olemassa funktio-osoittimesta toiseen. Esimerkiksi:

```
int calc( int, int );
int (*pfi2s)( const string &, const string & ) = 0;
int (*pfi2i)( int, int ) = 0;

int main() {
    pfi2i = calc;  // ok
    pfi2s = calc;  // virhe: tyypit eivät täsmää
    pfi2s = pfi2i; // virhe: tyypit eivät täsmää
    return 0;
}
```

Funktio-osoitin voidaan alustaa tai siihen voidaan sijoittaa nolla-arvo, joka ilmaisee, että osoitin ei osoita mihinkään funktioon.

### 7.9.3 Käynnistys

Osoitinta funktioon voidaan käyttää funktion kutsumiseen, johon se viittaa. Käänteisoperaattoria ei tarvita funktion käynnistykseen. Sekä suora funktion kutsu, jossa käytetään funktion nimeä, että epäsuora funktion kutsu, jossa käytetään osoitinta, voidaan kirjoittaa samalla tavalla. Esimerkiksi:

```
#include <iostream>

int min( int*, int );
int (*pf)( int*, int ) = min;

const int iaSize = 5;
int ia[ iaSize ] = { 7, 4, 9, 2, 5 };

int main() {
    cout << "Direct call: min: "
        << min( ia, iaSize ) << endl;

    cout << "Indirect call: min: "
        << pf( ia, iaSize ) << endl;

    return 0;
}

int min( int* ia, int sz ) {
    int minVal = ia[ 0 ];
    for ( int ix = 1; ix < sz; ++ix )
        if ( minVal > ia[ ix ] )
            minVal = ia[ ix ];
    return minVal;
}
```

Kutsu

```
pf( ia, iaSize );
```

voidaan kirjoittaa myös pitempää eksplisiittistä osoittimen ilmaisua käyttäen:

```
(*pf)( ia, iaSize );
```

Molemmat muodot johtavat samaan tulokseen, vaikkakin jälkimmäinen on selkeämpi käyttäjälle ja ilmaisee, että kutsu tehdään funktio-osoittimen kautta.

Tietysti, jos funktio-osoittimen arvo on nolla, molemmat käynnistykset johtavat suorituksenaikaiseen virheeseen. Vain sellaisia osoittimia, jotka on alustettu tai joihin on sijoitettu viittaus funktioon, voidaan turvallisesti käyttää funktion käynnistykseen.

### 7.9.4 Osoitintaulukot funktioihin

On mahdollista esitellä osoitintaulukoita funktioihin. Esimerkiksi

```
int (*testCases[10])();
```

esittelee testCases:in taulukoksi, jossa on kymmenen elementtiä. Jokainen elementti on osoitin funktioon, joka saa kaksi argumenttia ja jonka paluutyyppi on int.

Esittelyitä kuten tapauksessa testCases on vaikea lukea, koska on vaikea tulkita, mihin osaan esittelystä funktiotyyppi liittyy. Sellaisissa tapauksissa typedef-nimen käyttö saa esittelyt huomattavasti helpommiksi lukea. Esimerkiksi:

```
// typedef-nimet saavat esittelyt helpommiksi lukea
typedef int (*PFV)(); // typedef osoittimelle funktiotyyppiin

PFV testCases[10];
```

Tämä testCases:in esittely on samanarvoinen edellisen kanssa.

Funktion kutsu, johon yksi testCases:in elementeistä viittaa, voisi näyttää tältä:

```
const int size = 10;
PFV testCases[size];
int testResults[size];

void runtests() {
    for ( int i = 0; i < size; ++i )
        // taulukon elementin kutsuminen
        testResults[ i ] = testCases[ i ]();
}
```

Osoitintaulukko funktioihin voidaan alustaa alkuarvoluettelolla, jossa jokainen alkuarvo edustaa samantyyppistä funktiota kuin taulukon elementin tyyppi. Esimerkiksi:

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

typedef int ( *PFI2S )( const string &, const string & );
PFI2S compareFuncs[2] =
{
    lexicoCompare,
    sizeCompare
};
```

Osoitin voidaan esitellä myös compareFuncs:iin. Sellainen osoitin on tyyppiä “osoitin osoitin-taulukkoon, jonka elementit osoittavat funktioihin”. Määrittely näyttää tältä:

```
PFI2S (*pfCompare)[2] = compareFuncs;
```

Esittely voidaan jakaa osiin seuraavasti:

```
(*pfCompare)
```

Käänteisoperaattori esittelee pfCompare:n osoittimeksi. Sen jälkeen tuleva [2] ilmaisee, että pfCompare on osoitin kahden elementin taulukkoon:

```
(*pfCompare)[2]
```

Typedef-nimi PFI2S ilmaisee taulukon elementin tyypin, joka on "osoitin funktioon, joka palauttaa tyypin int ja saa kaksi const string & -tyyppistä parametria". Taulukon elementin tyyppi on sama kuin lausekkeen &lexicoCompare tyyppi. Se on myös samanlainen kuin compareFuncs:in ensimmäisen elementin tyyppi, joka on voitu hankkia kirjoittamalla jommallakummalla tavalla seuraavista:

```
compareFunc[ 0 ];
(*pfCompare)[ 0 ];
```

Jotta ohjelmoija voi kutsua lexicoCompare()-funktiota pfCompare-osoittimen kautta, hänen pitää kirjoittaa jommallakummalla tavalla seuraavista:

```
// samanarvoiset käynnistykset
pfCompare[ 0 ]( string1, string2 );      // lyhennetty
((*pfCompare)[ 0 ])( string1, string2 ); // täsmällinen
```

### 7.9.5 Parametrit ja paluutyypit

Palaan takaisin ongelmaan, joka esitettiin tämän kohdan alussa. Meille annettiin tehtäväksi kirjoittaa lajittelufunktio. Kuinka osoittimia funktioihin voidaan käyttää tämän funktion kirjoittamiseen? Koska funktioparametri voi olla osoitin funktioon, voimme välittää osoittimen funktioon argumenttina lajittelufunktiolle ilmaistaksemme, mitä vertailuoperaatiota sen tulee käyttää:

```
int sort( string*, string*,
      int (*)( const string &, const string & ) );
```

Jälleen: typedef-nimen käyttö saa sort()-funktion esittelyn helpommaksi lukea:

```
// typedef saa sort()-funktion esittelyn helpommaksi lukea
typedef int ( *PFI2S )( const string &, const string & );
int sort( string*, string*, PFI2S );
```

Koska funktio, jota tullaan käyttämään useimmiten, on lexicoCompare(), voidaan osoitin funktioon -parametria käyttää oletusargumenttina:

```
// kolmannelle parametrille annetaan oletusargumentti
int lexicoCompare( const string &, const string & );
int sort( string*, string*, PFI2S = lexicoCompare );
```

sort()-funktion määrittely voisi näyttää tältä:

```
1 void sort( string *s1, string *s2,
2          PFI2S compare = lexicoCompare )
3 {
4    // rekursion pysäytysehto
5    if ( s1 < s2 ) {
```

```
6       string elem = *s1;
7       string *low = s1;
8       string *high = s2 + 1;
9
10      for (;;) {
11          while ( compare( *++low, elem ) < 0 && low < s2) ;
12          while ( compare( elem, *--high ) < 0  && high > s1) ;
13
14          if ( low < high )
15              low->swap(*high);
16          else break;
17      }  // for(;;) loppuu
18
19      s1->swap(*high);
20      sort( s1, high - 1 );
21      sort( high + 1, s2 );
22    } // if ( s1 < s2 ) loppuu
23 }
```

sort() on toteutus C. A. R. Hoaren *quicksort*-algoritmista. Katsokaamme funktion määrittelyä tarkemmin. Funktio lajittelee taulukon elementit väliltä s1 ja s2. sort() on rekursiivinen funktio, joka käyttää itseään vähitellen pienempiin alitaulukoihin. Pysäytysehto toteutuu, kun s1 viittaa samaan elementtiin kuin s2 tai elementtiin, joka on s2:n viittaamaa elementtiä suurempi (rivi 5).

Elementtiä elem (rivi 6) sanotaan *osiointielementiksi*. Kaikki elementit, jotka ovat aakkosissa pienempiä kuin elem, siirretään elem-elementin vasemmalle puolelle; kaikki suuremmat elementit oikealle. Taulukko on nyt osioitu kahteen alitaulukkoon. sort()-funktiota kutsutaan rekursiivisesti kummallekin (rivit 20 ja 21).

for(;;)-silmukan tarkoitus on suorittaa osiointi (rivit 10 – 17). Silmukan jokaisella toistokerralla low kasvatetaan indeksoimaan sen taulukon ensimmäistä elementtiä, joka on suurempi tai yhtäsuuri kuin elem (rivi 11). Samalla tavalla high vähennetään indeksoimaan sen taulukon oikeanpuoleisinta elementtiä, joka on yhtäsuuri tai pienempi kuin elem (rivi 12). Jos low ei ole enää pienempi kuin high, elementit on osioitu ja pysäytämme silmukan; muussa tapauksessa elementit on vaihdettu keskenään ja seuraavat silmukan toistokerrat alkavat (rivit 14–16). Vaikka taulukko on osioitu, elem on silti taulukon ensimmäinen elementti. swap() sijoittaa rivillä 19 elem-elementin lopulliseen paikkaansa taulukkoon ennen kuin sort()-funktiota käytetään kahteen alitaulukkoon.

Taulukkoelementtien vertailu tehdään kutsumalla funktiota, johon compare viittaa (rivit 11 – 12). swap()-merkkijono-operaatiota kutsutaan, kun halutaan vaihtaa keskenään merkkijonot, joihin taulukon elementit viittaavat. (swap()-merkkijono-operaatio esiteltiin kohdassa 6.11.)

Seuraavassa main()-funktion toteutuksessa käytetään lajittelufunktiotamme:

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

```
// normaalisti näiden tulisi olla otsikkotiedostossa
int lexicoCompare( const string &, const string & );
int sizeCompare( const string &, const string & );
typedef int (*PFI)( const string &, const string & );
void sort( string *, string *, PFI=lexicoCompare );

string as[10] = { "a", "light", "drizzle", "was", "falling",
                  "when", "they", "left", "the", "museum" };

int main() {
    // kutsu sort()-funktiota, joka käyttää oletusargumenttia vertailuun
    sort( as, as + sizeof(as)/sizeof(as[0]) - 1 );

    // näytä lajitellun taulukon tulos
    for ( int i = 0; i < sizeof(as)/sizeof(as[0]); ++i )
        cout << as[ i ].c_str() << "\n\t";
}
```

Kun ohjelma käännetään ja suoritetaan, tuottaa ohjelma seuraavan tulostuksen:

```
"a"
"drizzle"
"falling"
"left"
"light"
"museum"
"the"
"they"
"was"
"when"
```

Funktion parametri ei voi olla funktiotyyppi. Sen sijaan funktiotyyppinen parametri konvertoidaan automaattisesti osoitin funktioon -tyypiksi. Esimerkiksi:

```
// typedef edustaa funktiotyyppiä
typedef int functype( const string &, const string & );
void sort( string *, string *, functype );
```

Kääntäjä kohtelee `sort()`-funktiota kuin se olisi esitelty näin

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

Kaksi edellistä `sort()`-funktion esittelyä ovat samanlaisia.

Huomaa, että sen lisäksi, että osoitinta funktioon voidaan käyttää parametrityyppinä, se voi olla funktion paluuarvona. Esimerkiksi:

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

Tämä esittelee `ff()`-funktion, joka saa yhden `int`-tyyppisen parametrin. Se palauttaa osoittimen funktioon, jonka tyyppi on

```
int (*)( int*, int );
```

Jälleen: typedef-nimen käyttö saa esittelyt huomattavasti helpommiksi lukea. Esimerkiksi typedef-nimi PF saa tulkinnan helpommaksi, että ff()-funktiolla on paluutyyppi, joka on osoitin funktioon:

```
// typedef-nimi saa esittelyt helpommiksi  lukea
typedef int (*PF)( int*, int );

PF ff( int );
```

Funktiota ei voi esitellä niin, että se palauttaisi funktiotyypin. Jos tehtäisiin niin, saisi se aikaan käännöksenaikaisen virheilmoituksen. Esimerkiksi ff()-funktiota ei olisi voitu esitellä seuraavasti:

```
// typedef edustaa funktiotyyppiä
typedef int func( int*, int );

func ff( int ); // virhe: ff()-funktion paluutyyppi on funktiotyyppi
```

### 7.9.6 Osoittimet extern "C" -funktioihin

On mahdollista esitellä osoittimia funktioihin, jotka viittaavat muilla ohjelmointikielillä kirjoitettuihin funktioihin. Se voidaan tehdä linkitysdirektiivejä käyttäen. Esimerkiksi pf-osoitin viittaa C-funktioon:

```
extern "C" void (*pf)(int);
```

Kun pf-osoitinta käytetään funktion kutsumiseen, on kutsuttu funktio C-funktio.

```
extern "C" void exit(int);

// pf viittaa C-funktioon exit()
extern "C" void (*pf)(int) = exit;

int main() {
    // ...
    // kutsuu C-funktiota, nimittäin exit()-funktiota
    (*pf)(99);
}
```

Osoittimella C-funktioon ei ole samaa tyyppiä kuin osoittimella C++-funktioon. Muista, että funktio-osoittimen alustus tai siihen sijoitus on sallittu vain, jos sijoitetun osoittimen tyyppi on täsmälleen samanlainen kuin sijoitusoperaattorin oikealla puolella olevan funktion tai osoittimen tyyppi. Sen vuoksi osoitinta C-funktioon ei voida alustaa tai siihen sijoittaa niin, että se osoittaisi C++-funktioon (ja päinvastoin). Kun sellainen epätäsmällisyys löytyy, annetaan siitä käännöksen aikana virheilmoitus. Esimerkiksi:

```
void (*pf1)(int);
extern "C" void (*pf2)(int);
int main() {
    pf1 = pf2; // virhe: pf1 ja pf2 ovat tyypeiltään erilaisia
```

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

Huomaa, että joissakin C++-toteutuksissa osoittimen piirteet C-funktioon ovat samoja kuin osoittimen piirteet C++-funktioon. Jotkut kääntäjät voivat sallia edellisen sijoituksen kielilaajennuksena.

Kun esittelyyn käytetään linkitysdirektiiviä, se vaikuttaa kaikkiin tuon esittelyn funktioesittelyihin. Seuraavassa esimerkissä pfParm-parametri on myös osoitin C-funktioon. Linkitysdirektiivi vaikuttaa funktioon, johon tämä parametri viittaa.

```
// pfParm on osoitin C-funktioon
extern "C" void f1( void(*pfParm)(int) );
```

f1() on sen vuoksi C-funktio, jolla on yksi parametri, joka on osoitin C-funktioon. Jälleen, koska osoittimella C-funktioon ei ole sama tyyppi kuin osoittimella C++-funktioon, argumentin, joka välitetään f1()-funktiolle, pitää olla C-funktion nimi tai osoitin C-funktioon. (Jälleen toteutuksissa, joissa osoittimille C-funktioihin ja osoittimilla C++-funktioihin on samanlaiset piirteet, kääntäjä voi tukea kielilaajennusta, joka sallii osoittimen C++-funktioon välitettävän f1()-funktiolle argumenttina.)

Koska linkitysdirektiiviä käytetään esittelyn kaikkiin funktioihin, kuinka voidaan esitellä parametri C++-funktiolle, joka on osoitin C-funktioon? Ratkaisu on typedef-nimen käyttö. Esimerkiksi:

```
// FC edustaa tyyppiä:
// C-funktio saa yhden int-parametrin ja palauttaa void-tyypin
extern "C" typedef void FC( int );

// f2() on C++-funktio, jolla on parametri,
// joka on osoitin C-funktioon
void f2( FC *pfParm );
```

### Harjoitus 7.21

Kohdassa 7.5 määritellään funktio factorial(). Määrittele osoitin funktioon, joka voi osoittaa factorial()-funktioon. Käynnistä funktio tämän osoittimen avulla ja generoi numeron 11 kertoma.

### Harjoitus 7.22

Mitä ovat seuraavien esittelyiden tyypit?

```
(a) int (*mpf)(vector<int>&);
(b) void (*apf[20])(double);
(c) void (*(*papf)[2])(int);
```

Kuinka käyttäisit typedef-nimiä saadaksesi esittelyt helpommin luettaviksi?

### Harjoitus 7.23

Seuraavat ovat C-kirjaston funktioita, jotka on määritelty <cmath>-otsikkotiedostoon:

```
double abs(double);
double sin(double);
double cos(double);
double sqrt(double);
```

Kuinka esittelisit osoitintaulukon C-funktioihin ja alustaisit taulukon niin, että se sisältäisi nuo neljä funktiota? Kirjoita main()-funktio, joka kutsuu sqrt()-funktiota taulukon elementin kautta argumentilla 97.9.

### Harjoitus 7.24

Palatkaamme takaisin sort()-esimerkkiin. Laadi määrittely funktiolle

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

niin, että jos kaksi parametria viittaavat samankokoisiin merkkijonoihin, silloin sizeCompare() palauttaa arvon 0; muussa tapauksessa, jos ensimmäisen parametrin merkkijono on lyhempi kuin toisen parametrin, palautetaan negatiivinen numero; jos se on suurempi, palautetaan positiivinen numero. Muista, että size()-merkkijono-operaatio antaa merkkijonon koon. Muuta main()-funktiota kutsumaan sort()-funktiota niin, että sen kolmas argumentti on osoitin sizeCompare()-funktioon.